AD-A063 900 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN--ETC(U)
SEP 78 S K SHASTRY N00014-76-C-0416
UNCLASSIFIED 78-01 NL

1 OF 6
AD
A063900

LEGES SINE MORIBUS VANAE

UNIVERSITY of PENNSYLVANIA

The Moore School of Electrical Engineering

PHILADELPHIA, PENNSYLVANIA 19104

Automatic Program Generation Project
Department of Computer and Information Science
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pa. 19174.

2

Technical Report

VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN AUTOMATIC GENERATION OF PROGRAMS

by

Subramanya K. Shastry

Prepared For
Information Systems Program
Office of Naval Research
Arlington, Virginia 22217

Under Contract N00014-76-C-0416

September 15, 1978

Moore School Report #78-01

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

2 Department of Computer and Information Science ✓
The Moore School of Electrical Engineering (D2) ✓
University of Pennsylvania
Philadelphia, Pennsylvania 19104

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE

6 Verification And Correction Of Non-Procedural Specifications In Automatic Generation of Programs,

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

9 Technical Report, September 15, 1978

5. AUTHOR(S) *(First name, middle initial, last name)*

10 Subramanya K. Shastry

11

6. REPORT DATE

15 Sep[redacted] 1978

12 553p.

7a. TOTAL NO. OF PAGES
560

7b. NO. OF REFS
14 80

8a. CONTRACT OR GRANT NO.
15 N00014-76-C-0416 new

b. PROJECT NO.
NR-049-153

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
Moore School Report 78-01

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT

Reproduction in whole or in part permitted for purposes of the United States Government.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY
Information Systems Program
Office of Naval Research
U.S. Navy

13. ABSTRACT

This dissertation deals with the design and implementation of an interactive Automatic Program Generator, which will generate PL/1 programs from a non-procedural description of a problem in MODEL (MOdule DEscription Language). MODEL has many characteristics of very high level languages, being at the same time, descriptive. MODEL processor is quite tolerant of errors in a user specification (like incompleteness, inconsistencies, and ambiguities), and resolves those errors, either by generating additional statements, or by updating the user supplied statements.

A special form of graph called, "Array Graph", is used to represent the MODEL specification. Informally, an array graph is a compact representation of a conventional directed graph. Conventional graph algorithms are applied directly to the array graphs, and in particular, a necessary and sufficient condition for "sequenceability" of array graphs is derived.

Array graph representation has been found quite useful in analyzing MODEL specifications consisting of iterations and recursions.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

410 039

Unclassified
Security Classification

A-31408

16

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
automatic program generation						
automatic programming						
array graphs						
correctness						
graph theory applications						
Module Description Language (MODEL)						
non-procedural language						
program generation						
program specification						
Schedulability						
Sequenceability						
Verification						
very high level language						

DD FORM 1 NOV 65 1473 (BACK)
S/N 0101-807-6821

Unclassified
Security Classification

A-31409

VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN AUTOMATIC GENERATION OF PROGRAMS

Subramanya K. Shastry

A DISSERTATION

in

Computer and Information Science

Presented to the Graduate Faculty of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

1978

Noah S Prywes

Supervisor of Dissertation

[illegible]

Graduate Group Chairman

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to my supervisor, Prof. N. S. Prywes, for introducing the topic and for his advice, direction and encouragement throughout the duration of research. Thanks are also due to Prof. A. Pnueli for his most helpful suggestions, especially in developing the theory on array graphs.

Thanks are due to my dissertation committee: Prof. P. Buneman, Prof. M. Franklin, Prof. H. L. Morgan, and Prof. A. Pnueli, for their helpful suggestions and to Profs. S. Gorn, A. Joshi and D. Lefkovitz for their suggestions during the development of the proposal.

The other members of the Automatic Programming Project, in particular: Yung Chang, Jorge Gana, Amihai Motro, Alex Pelin, Cihan Tinaztepe and Joe Weiss, have helped a great deal throughout the research for which I am grateful. Thanks are due to John Gordon and Ellen Sher for writing several subroutines for the second phase of the MODEL Processor (Summer 1977).

I would also like to thank Seth Bergmann and Richard Epstein for their useful suggestions and encouragement during the course of research.

I would like to thank Barbara Collins for her help in typing various technical reports and to the members of the

Moore School Computer Facility for their help in various forms.

My thanks are also due to the Office of Naval Research for the grant under which my research was done.

Finally, for the support, encouragement and help I needed to make this through, I thank especially, my wife, Appi.

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

<u>TABLE OF CONTENTS</u> <u>PAGE</u>

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

TABLE OF CONTENTS PAGE

List of Tables

List of Figures

List of Theorems

List of Algorithms

ABSTRACT

VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN AUTOMATIC GENERATION OF PROGRAMS

by S. K. Shastry

Supervisor: Prof. N. S. Prywes

This dissertation deals with the design and implementation of an interactive Automatic Program Generator, which will generate PL/1 programs from a non-procedural description of a problem in MODEL (MOdule DEscription Language). MODEL has many characteristics of very high level languages, being at the same time, descriptive. MODEL processor is quite tolerant of errors in a user specification (like incompleteness, inconsistencies, and ambiguities), and resolves those errors, either by generating additional statements, or by updating the user supplied statements.

A special form of graph called, "Array Graph", is used to represent the MODEL specification. Informally, an array graph is a compact representation of a conventional directed graph. Conventional graph algorithms are applied directly to the array graphs, and in particular, a necessary and sufficient condition for "sequenceability" of array graphs is derived.

Array graph representation has been found quite useful in analyzing MODEL specifications consisting of iterations and recursions.

CHAPTER 1

INTRODUCTION

1.1 Problem of Automatic Programming.

For a long time it has been realized that technological advances in hardware have far outstripped those in software. In addition, because each software must be individually handcrafted by a programmer, the cost of software is rising. To reduce this problem, several attempts have been made to automate software systems. As a result a variety of compilers, higher level languages, automated aids, etc., have emerged. These tools have served as a useful aid to the programmers, but are not enough to simplify the software production problems of today.

The research reported in this dissertation is aimed at solving the rising problems of software development by automating some phases of the software development process. This is achieved by developing a simple man-machine interface, in which the user describes a problem in a simple descriptive language, without direct reference to a particular computational technique, or a particular data operation. The interface, in turn, generates a program for the problem in a high level language, after resolving the errors in the user specification. This interface is quite different from the conventional language processors, because, the user in this case, need not be a programmer, and his specification of a problem could be completely dis-organized. The need to think of a problem as a sequence of actions and controls is eliminated, thus reducing the time and effort required by the user (or a programmer) to solve a problem.

The language used for the purpose is called MODEL, and it resembles more a mathematical language of functional and relational expressions. A problem description in MODEL almost follows the rule of Donald Michie (MIC68), that is:

> "If a passage of text is respectable mathematics, in generally accepted notation, then it will compile."

It is the function of the MODEL processor to generate the required algorithm (or procedure) from the non-procedural specification of a problem in MODEL.

O'Donnel (ODO77) points out that:

> "Descriptive languages are not as popular for practical computation as procedural languages, partly because of the efficient implementation of such languages may be very difficult. ..."

However, the MODEL Processor described in this dissertation shows how an efficient non-procedural language processor can be developed.

1.2 Summary of Research.

The ultimate objective of the research reported in this dissertation is the automation of the software development process. A broad overview of such an automated system is shown in Figure 1.1. It illustrates the differnt interactions between the Automatic Program Generater (MODEL Processor) and the user. The different parts of the MODEL processor, shown in the Figure, will be described later in Chapter 3. For the purpose of identifying the scope of this dissertation, another overview of the MODEL Processor is shown in Figure 1.2. In this figure, the Processor is divided into two parts. This dissertion describes the design and implementation of the first part (namely, Analysis of MODEL specification) and the different user interfaces (see also, the Figure 1.1). The second part of the MODEL processor is not yet implemented; however, the techniques used in the earlier version of the MODEL Processor (PRY76A) can be used in the design of this part of the system.

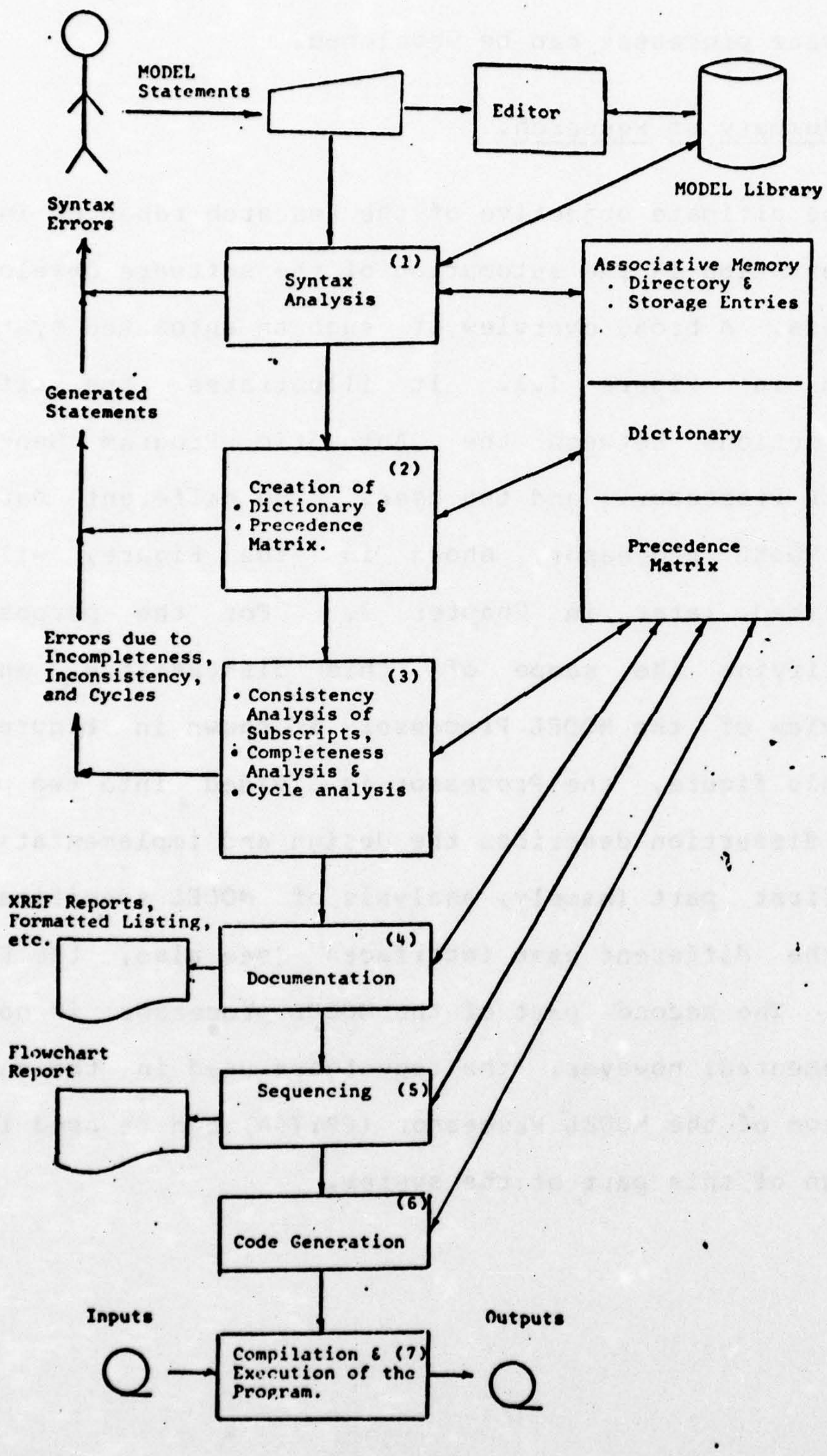


Figure 1.1. Overview of MODEL Processor.

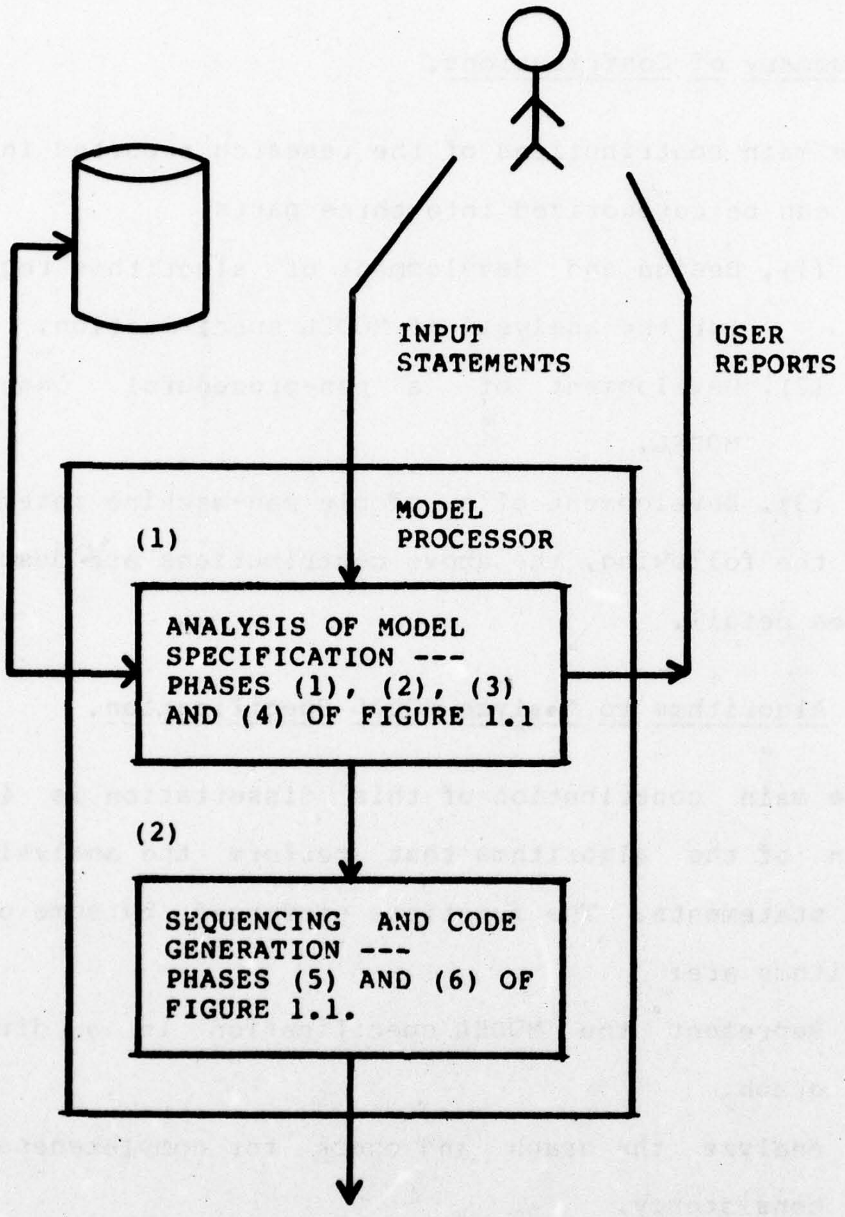

INPUT STATEMENTS
USER REPORTS
MODEL PROCESSOR
(1)
ANALYSIS OF MODEL SPECIFICATION --- PHASES (1), (2), (3) AND (4) OF FIGURE 1.1
(2)
SEQUENCING AND CODE GENERATION --- PHASES (5) AND (6) OF FIGURE 1.1.
PL/1 PROGRAM

Figure 1.2. Overview of the MODEL Processor.

1.3 Summary of Contributions.

The main contributions of the research reported in this paper can be categorized into three parts:

(1). Design and development of algorithms required for the analysis of MODEL specification,

(2). Development of a non-procedural language, MODEL,

and (3). Development of a simple man-machine interface.

In the following, the above contributions are described in more detail.

1.3.1 Algorithms to Analyze MODEL Specification.

The main contribution of this dissertation is in the design of the algorithms that perform the analysis of MODEL statements. The functions performed by some of the algorithms are:

(1). Represent the MODEL specification in a directed graph.

(2). Analyze the graph and check for completeness and consistency.

(3). Analyze every iterative computation, and check for consistency in the scope of iterations and in the specification of subscripts.

(4). Analyze every recursive relation and check for proper initiation and termination of the recursion.

(5). Resolve the errors detected during the execution of steps (2), (3) and (4) above, either by generating additional statements, or by updating the user supplied statements.

(6). Report the errors encountered during the analysis and the ways those errors were resolved.

(7). Generate user documentation, by producing various listings, cross reference report, and so on.

The novel feature of the MODEL Processor is the representation of the input specification in a compact form of directed graph, called array graph, and then apply the conventional graph algorithms directly to it. Such representation has been found quite useful in analyzing the completeness and consistency of iterative and recursive relations. Many other features of the specification can be exploited (parallelism, as an example) from this simple representation.

Some quite useful results on the analysis of array graphs have been derived in Chapter 9. These include (1) a necessary and sufficient condition for the "sequenceability" of array graphs, (2) algorithms for splitting (or decomposing) array graphs, and (3) a condition for the "schedulability" of array graphs.

1.3.2 The Non-procedural Language, MODEL.

Unlike conventional languages, MODEL is a non-procedural or a descriptive language. Some of the important characteristics of the language are:

(1). Non-procedural,

(2). Tolerant,

(3). Simpler, and

(4). More powerful.

The non-proceduralness of the language is quite important. It makes the task of the user very easy in describing a problem. He does not have to think of a problem in terms of processes. He regards the computer as a black box and his description of a problem is in terms of data or documents used as input or output, as well as formulae that describe the relationships between the data.

MODEL is a formal language, and uses some reserved words and punctuation common to programming languages. However, MODEL Processor is quite tolerant of many omissions and errors in the input statements by either generating new statements or by updating the user supplied statements. The MODEL Processor automatically resolves most of the errors in the user specification.

MODEL is simple to use. It consists essentially of two types of statements: data description statements and assertions. Data description statements define the data items used in the specification, and assertions define the relationships between different data items. MODEL does not have control statements like, DO, GOTO, etc., nor does it have the I/O operations like: READ, WRITE, and so on. All the required control statements and I/O operations are generated by the MODEL Processor. Most important of all, because of the non-procedural nature of the language, the user does not have to specify any sequence of operations.

Even though MODEL has a simple structure, it is quite powerful. In fact, in some cases, it is more powerful than the conventional languages. User can specify array operations as in higher level languages like APL. He can specify iterative and recursive relations. He can also specify complex data structures, similar to, but more powerful than the "REFER" option of PL/1.

Some of the characteristics of MODEL are common with the characteristics of the languages used in the earlier versions of MODEL Processor (see RIN76 and PRY76A). The main differences between the different versions are described in Chapter 2.

1.3.3 Man-Machine Interface.

The important requirement in designing a non-procedural language processor is the need of a simple interface, that analyzes the mathematical semantics of the language. therefore, if a relation between two data items is specified using an assertion, it may be possible to obtain a procedure (or a sequence of computations -- programming semantics) required to evaluate the item being defined. If a user has not supplied enough information, then there may be a way to generate that information. Any error in the input specification must be resolved and the user must be advised of such automatic resolution. Every input statement must be checked to see if it fits properly in the global environment, and there must be a way to sequence the different statements so as to obtain a procedural description of the problem. The MODEL Processor performs the above functions and can be regarded as a useful tool even for non-programmers to develop quite sophisticated programs. As can be seen from the flowchart of the MODEL Processor in Figure 1.1, there is a lot of interaction between the user and the processor thru various reports and documentation. This interaction helps the user in understanding the functions of the generated program and in correcting his errors. It also helps the processor to understand the user specification correctly.

A true interactive man-machine interface should look as shown in Figure 1.3. As shown in the figure, the user communicates with the processor in simple "English-like" sentences, and a preprocessor (labelled EMI -- English to MODEL Interface) translates the input into MODEL statements. This preprocessor may also use a "domain-expert" to resolve some errors (like incompleteness) in the input using some domain dependent knowledge. Even though, such a preprocessor has been proposed (see PRY76B), it has not been implemented for the MODEL Processor.

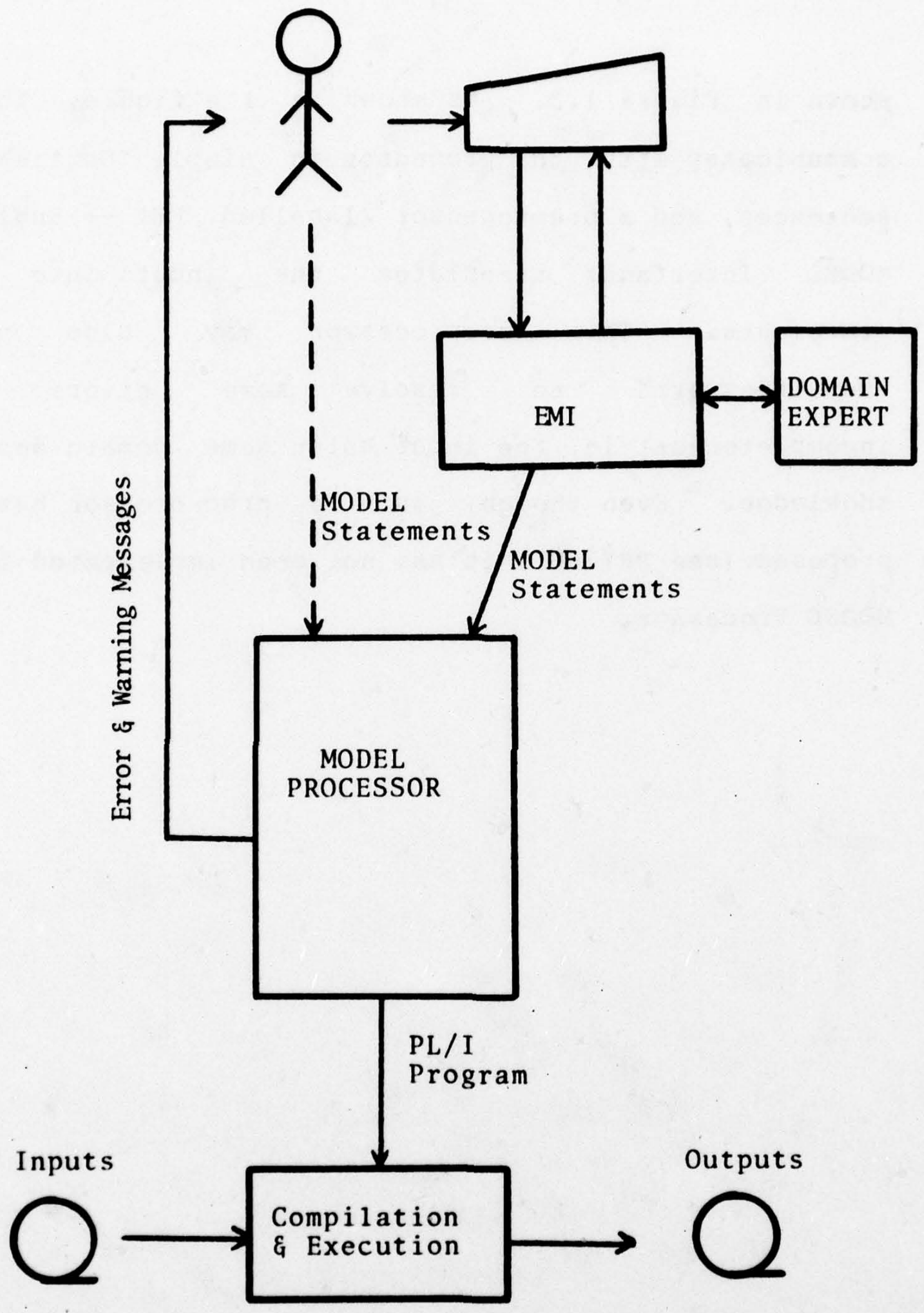


Figure 1.3. An Ideal Automatic Program Generator.

1.4 Organization of the Dissertation.

Chapter 2 gives the motivation for the research and a summary of other related work on Automatic Programming. Chapter 3 gives an overview of the MODEL Processor, briefly describing each of its phases.

Chapter 4 describes the MODEL language and can be used as a reference manual for MODEL. As an example, a simple Trasaction Processing problem is described in MODEL.

Chapter 5 describes Phase-1 of the MODEL Processor, namely the syntax analysis phase.

Chapter 6 describes the organization of the associative memory, dictionary, and the precedence matrix. It also describes some of the procedures that can be used to create and access the contents of the associative memory, dictionary and the precedence matrix.

Chapter 7 describes the organization of the MODEL statement database, and the procedures for creating and updating the database.

Chapter 8 describes Phase-2 of the MODEL Processor, the main function of which is to create the dictionary and the precedence matrix.

Chapter 9 describes Phase-3 of the MODEL Processor which performs the completeness and consistency analysis of the MODEL specification. The concept of array graph is also introduced in this chapter. Some theoretical results on array graphs are also presented in this chapter.

Chapter 10 describes Phase-4 of the MODEL processor which produces different user reports.

Chapter 11 gives an overview of the last three phases of the MODEL Processor which perform the sequencing, optimization and code generation, respectively. Also, it summarizes the conclusions that can be drawn based on the research, and suggested directions for further research.

CHAPTER 2

BACKGROUND, MOTIVATION, AND SUMMARY OF RELATED LITERATURE AND RESEARCH

2.1 Introduction.

Though present day computers are often called Automatic Computers, they are in fact far from being automatic. It is true that computers provide means for processing large and complex programs quickly and accurately. However, problem analysis and design, coding and testing precede the use of the computer. To make the problem representation easier, numerous computer languages have been developed. These languages are means of communication between the user and the computer. The main function of these languages is to translate user input statements to the machine instructions of the computer. Depending on the complexity of the language, each user statement may produce hundreds of machine language instructions. Thus, these high level languages relieve the user from the detailed description of a problem.

Despite the speed and accuracy of computers, the main factor in determining the usability of a computer is the cost of solving a problem. This cost includes:

(a). Cost of describing the problem in a language (this problem is usually handled by the application programmer who writes programs for the problem).

(b). Cost of formatting the input required for the execution of the program (this task is ususlly performed by the clerical staff who keypunch or type the input data into the system).

(c). Cost of executing the program or the cost of maintaining the system (this task is performed by the computer with some help from the programmer).

Figure 2.1 shows a simple problem solving process. In reality, the process is much more complicated than that shown in Figure 2.1. The computer may recognize many errors during the testing and execution phases of the problem solving process. Consequently error feedback loops are present in both these phases. This is illustrated in Figure 2.2. Output can be obtained only after the user modifies the problem description to eliminate the errors. Depending on the user proficiency in the language in which the problem is represented and in the computer system he uses, the above feedback process can continue for many iterations. Thus, the cost of programming may sometimes take a major share in the total cost of problem solving. This is because the user may not be well trained both in the field in which the problem is

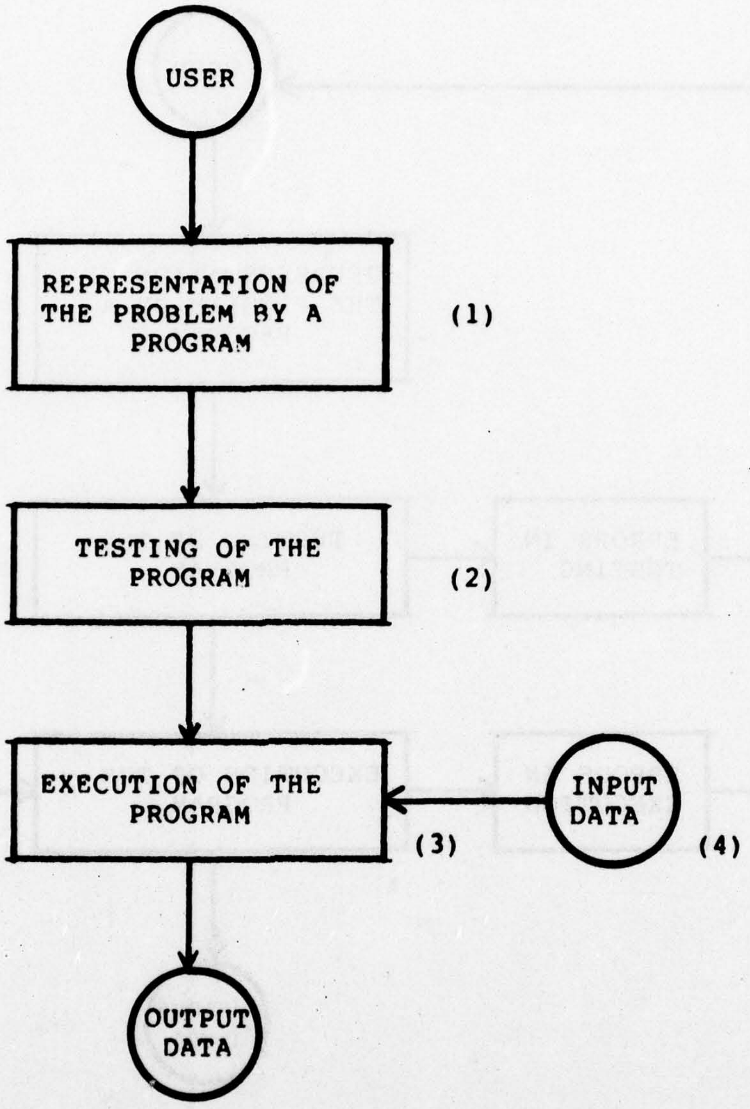


FIGURE 2.1. A Simple Problem Solving Process.

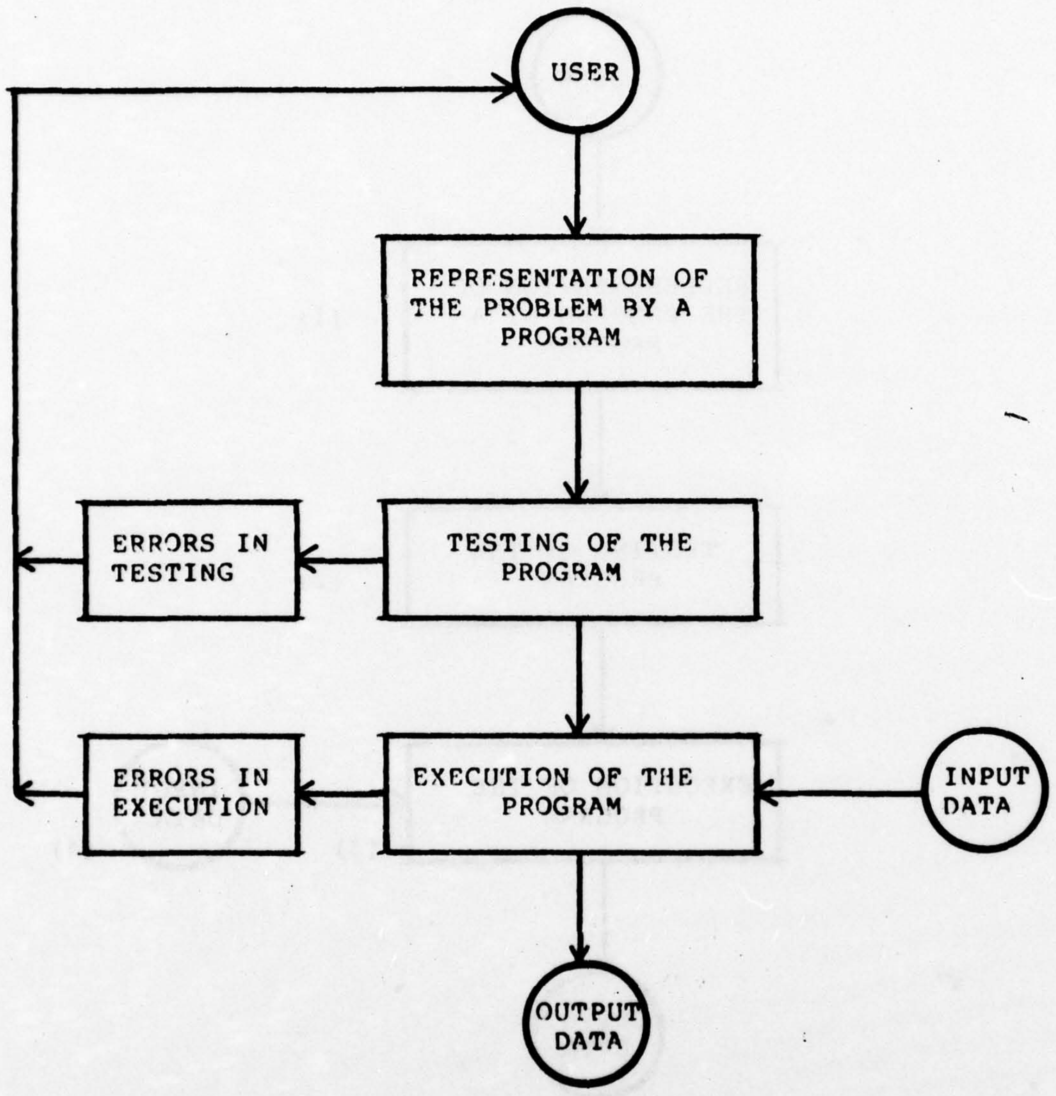


Figure 2.2. A Problem Solving Process with Error Feedback Loops.

to be solved and in the computer language in which the problem is to be represented.

2.1.1 <u>Software Difficulties in the use of Computers</u>.

As indicated in the earlier section, even an experienced programmer may spend a lot of time in debugging a program (that is, in resolving the errors caused during the testing and execution phases). Usually, most time is spent in resolving the errors in the execution phase. This is because present day language processors are not intelligent enough to analyze the program and indicate any faults in the program logic. The programmer loses control over the program when it increases in size and complexity, because, he can concentrate only on a small portion of the program at a time and fails to find out the faults during the linking process of the different parts of the program. Though the recent structured programming techniques have eliminated some of these problems, the programmer must be well experienced to write a program with few errors. Unfortunately, once he is well trained in a language, he hardly has any opportunity to write programs. His subordinates will then go thru the same learning process all over again.

2.1.2 Prerequisite Training.

It is not sufficient just to have training in the language in which the problem is to be solved. The user must be familiar with the computer system he uses. Usually, a typical application program requires a huge amount of Input/Output activity between the data base and the computer. The cost of I/O operations in executing a program can take a substantial share of the total cost. Therefore, the programmer must be familiar with the organization of the computer system, so that he can use the most efficient methods to represent data files. There are some programming languages which tolerate the user's lack of knowledge about the system he uses by providing default attributes for the data files. However, I/O operations in such a system are quite expensive.

2.1.3 Proceduralness.

When a programmer attempts to use a computer to solve his problem, he must state explicitly, and without ambiguity, the algorithm to be used in the solution of the problem. This algorithm is the precise step by step method of the solution. Such an algorithm must have several important characteristics: (1) Finiteness - the algorithm must end after a finite number of steps, (2) Definiteness - every step in the algorithm must be defined unambiguously, and (3) Generality - the algorithm must be applicable to a general type of problem rather than to a specific problem.

In the existing languages there is no facility to check if an algorithm has the above characteristics. Each step of an algorithm must be explicitly specified in a procedural language. So, incorrect specification of an algorithm causes errors in testing or execution phases of the problem solving process.

2.1.4 General Structure of a Software Development Process.

From the discussions of the previous sections, a software development process can be described by the flowchart outlined in Figure 2.3. The first phase, "determination of requirements and the cost effectiveness evaluation", is preliminary to the software development process and is concerned with the general objectives and pay-off of the development. The "system level specification" is usually done by an individual who is familiar with the problem for which the software is to be developed. The next three phases, namely, "partitioning of the system and data", "program generation" and "program maintenance" are done by the application programmer. If the cost of maintaining the resulting program is too high or the program performance is not satisfactory, then some decisions may have to be made to change the original specification. We are primarily interested in automating Phase (4) of the software development process depicted in Figure 2.3.

Many other overviews of the software development process are given in the literature. One detailed overview is given by Prywes (PRY77A, Figure 3). In this overview, basically, a software development process is divided into a top part and a bottom part. The top part performs the task of determining data processing requirements and generates a specification in a high level language acceptable by the bottom part.

In this dissertation I will be describing the design and implementation of the bottom part. An interface that performs the functions of the top part was proposed by Prywes (PRY76B) but is not yet implemented.

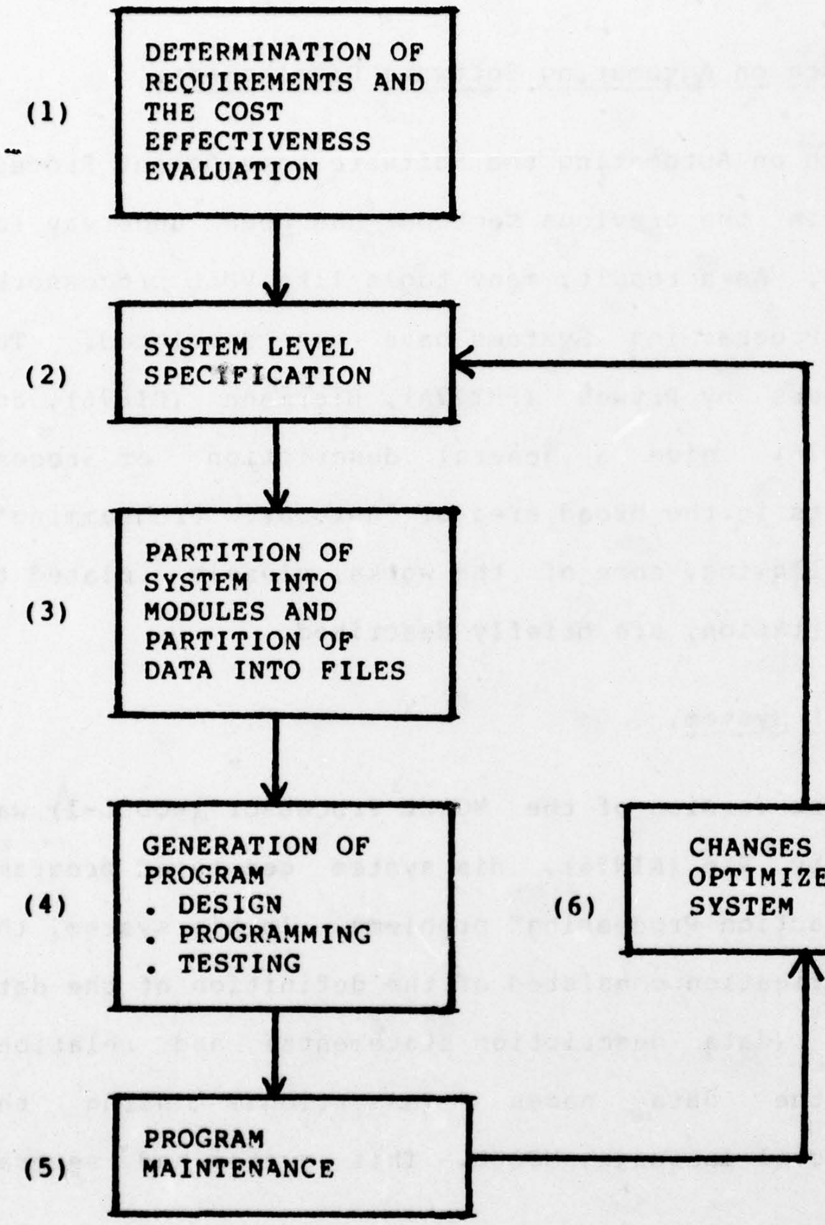

(1)
DETERMINATION OF REQUIREMENTS AND THE COST EFFECTIVENESS EVALUATION
(2)
SYSTEM LEVEL SPECIFICATION
(3)
PARTITION OF SYSTEM INTO MODULES AND PARTITION OF DATA INTO FILES
(4)
GENERATION OF PROGRAM
. DESIGN
. PROGRAMMING
. TESTING
(5)
PROGRAM MAINTENANCE
(6)
CHANGES TO OPTIMIZE THE SYSTEM

Figure 2.3. A Software Development Process.

2.2 Research on Automating Software Development.

Research on Automating the Software Development Process described in the previous section has been underway for many years. As a result, many tools like VHLL processors, Automatic Programming Systems have been developed. The review papers by Prywes (PRY77A), Biermann (BIE76), and Ruth (RUT76) give a general description of recent developments in the broad area of "Automatic Programming". In the following, some of the works, closely related to this dissertation, are briefly described.

2.2.1 MODEL System.

The first version of the MODEL Processor (MODEL-I) was developed by Rin (RIN76). His system generated programs for "Transaction Processing" problems. In his system, the user specification consisted of the definition of the data structures (data description statements) and relations between the data names (assertions) using the non-procedural language MODEL. This system had several drawbacks.

(1). The language was not too easy to use. The user had to explicitly define every variable used in the specification. The assertions used the syntax of PL/I, and were never scanned. Therefore, the user, again, had to explicitly specify every source and target varaible used in an assertion.

(2). The system was restricted to a certain class of problems. It could handle only one driving file, and one type of record in any file.

(3). Iterations could not be specified easily.

(4). Very few checks were performed for consistency and completeness of specification. The system did not attempt to resolve any error in the user specification.

Most of the above problems were resolved in the second version of the MODEL system (MODEL-II) system developed by the Automatic Program Generation Group at the Moore School (of which the author was a member, see also, PRY76A). In MODEL-II, the syntax of the language was simplified to a great extent, the allowable data structure was extended, and the user could specify the iterations fairly easily. These extensions eliminated the drawbacks (1), (2) and (3) of Rin's system (mentioned above). The above extensions were based on the specification language described by Pnueli (PNU76A, PNU77).

The MODEL system described in this dissertation (will be referred to as MODEL-III, or simply MODEL) is different from the earlier versions in the following sense:

(1). The language is simple, and the user is not required to specify every detail.

(2). The language is more powerful. The user can specify recursions and iterations easily. He can specify array operations without the use of subscripts (as in APL).

(3). It is quite tolerant of user errors. It resolves the errors by generating new statements or updating the user supplied statements.

(4). It uses a special form of directed graph to represent the MODEL specification and analyzes it for consistency and completeness, and then, sequences the different operations implied by it.

There are other extensions that are being built into the MODEL Processor. They are: (1) Adding the modelling and simulation techniques needed in natural and social sciences to the knowledge base (GAN78), (2) Partitioning of the software system progressively in a top-down manner into sub-modules (GIB75), and (3) Exploitation of parallelism in MODEL Programs (CHE78).

2.2.2 BDL <u>System</u>.

BDL (Business Definition Language) is an automatic programming system developed by IBM (GOL75A, GOL75B, HAM74, HAM77 and LEA77). Programming in BDL begins by drawing on a screen (of an interactive Graphics Display terminal) labelled boxes representing principal operating entities within the application area to be automated. These boxes are connected by paths indicating the lines of communication among them. The semantics associated with this part of BDL is that the step from which the path emanates is expected to produce a set of documents at some instant of time. Thus, the language is a data flow language with steps producing all their output simultaneously and executing whenever their input is ready.

Each box in the original BDL program can be described further using another BDL program. The user can continue this process till each box can no longer be sub-divided further. At this time he has to indicate the details of relation between the group of input documents and group of output documents. This is achieved by a tabular language. Part of this tabular program for a problem is shown in Figure 2.4. In this figure, the name suffixed by "'s" is a repeating data name. The causality for the name "Invoice" is given as "One PER LIKE Order's". LIKE

Order's are defined to be "Order's WITH COMMON Cust#".

Group/Field	Causality/ Derivation	Name	Definition
Invoice's	One PER Like Order's	Like Order's	Order's WITH COMMON Cust#
		Order's	INPUT

Figure 2.4

The flowchart specification and the tabular specification, together form a BDL specification for a problem. The flowchart specification of a problem in BDL may be quite easy for non-programmers, because, they do not have to learn the syntax and semantics of the formal specification problem. However, the tabular language has a lot of syntax built into it (for example, keywords like WITH, COMMON, LIKE and so on) and may be difficult for use by non-programmers.

2.2.3 <u>PSI System</u>.

The PSI system developed at Stanford (GRE76), uses as in BDL, a Model based approach. However, in this system provision is made for the incorporation of an independent domain expert module that "knows" the terminology of a particular domain and is familiar with the common activities. Thus, for different problem domains, just this domain expert has to be replaced, without any change to the rest of the system. The function of the domain expert is to free the user from making explicit descriptions and associations of things that are obvious to him.

2.2.4 USC/ISI System.

The APS (Automatic Programming System, BAL72, BAL74, HEI76) developed by the group at ISI is similar to the PSI system. However, the important difference is its emphasis on "domain independence". This means that the system is not primed with information about the specific problem area, but must obtain all of this information. The user-system dialog consists of the user initially stating his problem, from which the system constructs a "loose model". Then the system, through a process called "model completion", attempts to transform this loose model into an operational, interpretable form called "precise model". The model completion process usually requires further dialogue with the user.

2.2.5 SURGE System.

SURGE is a COBOL pre-processor (PET76) for the machine generation of source programs. The capabilities of SURGE are described as file sorting, selective retrieval and tabular report preparation, including multiple levels of totals and capability for certain other kinds of computation. It has however, a rigid syntax, and is not applicable to problem areas other than the ones mentioned above.

2.2.6 PROTOSYSTEM-I.

PROTOSYSTEM-I is developed by the Automatic Program Generation Group at MIT (RUT76). It consists of top and bottom parts. The top part consists of the man-machine interface and a knowledge base on business management of inventory, distribution, or procurement. It is intended to play the role of a management consultant for users with such business operation problems. The research on Program Writer, by Long (LON77), on questionaire approaches employed in consultant's role by Bosyj (BOS76) and Malhotra (MAL75) and on the OWL language used for the user communication by Martin (MAR74) are some of the works on the top part of the PROTOSYSTEM-I. The bottom part of PROTOSYSTEM-I was implemented by Ruth (RUT75, RUT76A) which obtains a data processing specification from the top part, performs system design, and generates PL/1 code.

The system specification language (SSL) used in the bottom part of PROTOSYSTEM-I somewhat resembles MODEL. For example, a relation in SSL looks like:

PAY IS HOURS_WORKED * RATE

Repetition inherent in the data items PAY, HOURS_WORKED are implicit here, just as in MODEL.

2.2.7 <u>System for Business Automation</u>.

The System for Business Automation (SBA) is developed at IBM (ZLO77b, PET77) and is a high level data base management language. A subset of SBA, called Query-BY-Example is also described in many reports (ZLO77A, ZLO75A, ZLO76 and ZLO75B). These languages provide provisions to query, update, define, and control a relational data base. In these systems, when the user performs an operation against the data base, he fills in an example of a solution to that operation in skeleton tables that can be associated with actual tables in the data base. As opposed to English-like query languages by which the user has to conform to the phrase structure of the language, the Query-by-example may enter any expression as an entry as long as it is syntactically correct.

2.2.8 LUCID.

LUCID (ASH77) is a formal system in which a non-procedural language, LUCID, is used to write programs, and at the same time verify them. A LUCID program can be thought of as a collection of commands describing an algorithm in terms of assignments and loops, but at the same time, LUCID is strictly a denotational language, and the statements of a LUCID program can be interpreted as true mathematical assertions about the results and effects of the program. As in PROTOSYSTEM-I, LUCID has the single assignment property, and each assignment in LUCID is equivalent to an equation. Even though, LUCID does not employ the concept of arrays (at least in the initial implementation), the "history" of a variable can be used to interpret an array. The operators like, "first", and "next", are used to define iteration, along with the operators like "as soon as" (like the keyword ENDGRP in MODEL) are used to terminate iterations.

Even though LUCID is claimed to be a non-procedural language, programs in LUCID do not look different from the iterative programs in conventional languages. The user has to think of loops and loop control variables when trying to write a LUCID program. For example, a LUCID program to compute a factorial can be written as follows:

```
N = first input;
first F = 1;
first I = 1;
next F = F * I;
next I = I + 1;
output = F as soon as I >= N;
```

which is no better than a procedural program in a conventional language.

2.2.9 <u>NOPAL System</u>.

The NOPAL system is developed at the Moore School, for designing functional and fault isolation tests of analog circuit equipment and for generating corresponding programs for computer controlled automatic test equipment (PRY75A). NOPAL system consists of two parts: the top part and the bottom part. The top part was developed by Tinaztepe (TIN77) and determines efficient tests and their respective diagnoses. The bottom part was developed by Chang (CHA 77) and accepts as input, the tests specified in the NOPAL language (CHE76) and produces as output, an efficient program in the OPAL high level test equipment programming language.

2.2.10 <u>ISDOS</u>.

The ISDOS (Information System Design and Optimization System) project at the University of Michigan has developed a documentation aid for composing and reporting system specifications (TEI72). It uses a Problem Statement Language (PSL) to express system functional specification formally. This system is a useful tool for a problem definer to express a problem without having to specify how the task will be carried out. This specification is analyzed by another system called Problem Statement Analyzer (PSA) which (1) performs consistency and completeness analysis and (2) performs dynamic analysis to indicate time dependent relationships of data. The result of these analyses is an error free problem statement in machine readable format. The second output is a coded statement for use in physical system design process.

CHAPTER 3

OVERVIEW OF THE MODEL PROCESSOR

In this chapter, the functions of the MODEL Processor are briefly described. The MODEL Processor can be divided into the following 7 phases:

1. Syntax Analysis.
2. Determination of Precedence Relationships.
3. Analysis of the Directed Graph.
4. Documentation.
5. Sequencing.
6. Code Generation.
7. Compilation and execution of the generated Program.

Figure 3.1 gives an overview of the MODEL Processor. In the following sections, the different phases of the MODEL processor are briefly described.

3.1 Syntax Analysis.

In this phase of the MODEL processor, the MODEL statements are solicited from the user, one at a time, and each statement is analyzed for proper syntax.

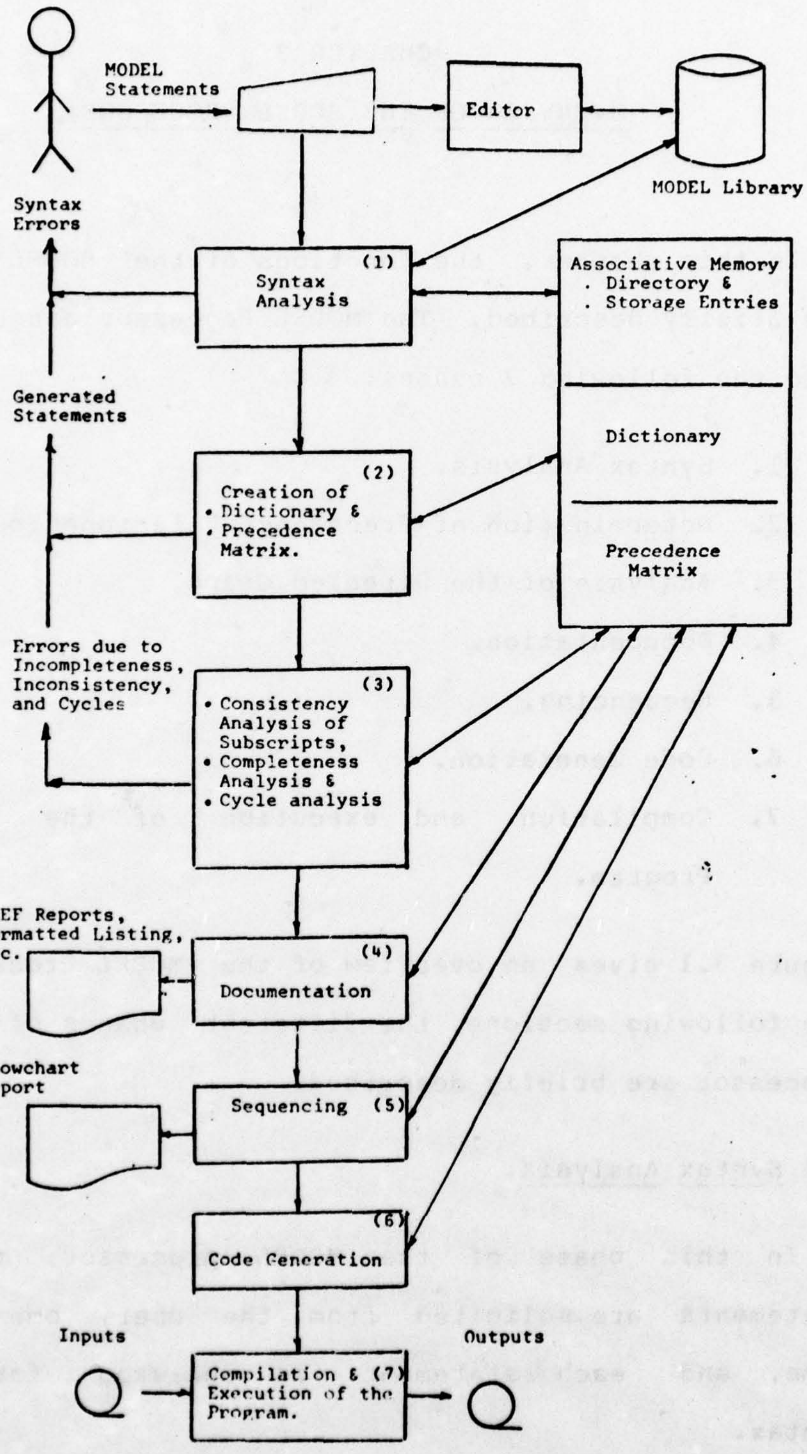


Figure 3.1. Overview of MODEL Processor.

Alternately, the user can use a conventional text editor to create a file of MODEL statements and save it in a secondary storage. The syntax analyzer will then obtain statements from the file, one at a time, and again, each statement is analyzed for proper syntax. Any error found during the syntax analysis is reported to the user. A novice can also interact with the processor to obtain the proper syntax of a particular statement, or to obtain an explanation of the error reported by the system. This phase is furter described in Chapter 5.

After successfully analyzing a MODEL statement for proper syntax, it is saved in the associative memory. Associative Memory, as shown in figure 3.1, consists of the following 4 parts:

1. Directory
2. Storage entry area
3. Dictionary

and 4. Precedence Matrix.

Directory consists of one entry for each name used by the user and one entry for each reserved key name used in the system. Storage entry area consists of a storage entry for each MODEL statement. Only the directory and the storage entries are created during the syntax analysis. The dictionary and the Precedence Matrix are created during the second phase of the MODEL processor (see also, section 3.2). However, the directory entries and the storage entries may be updated even during the second and third phases of the MODEL Processor, either by changing a user supplied statement or by generating additional statements to resolve some user errors. Organization of the Associative Memory is described in Chapter 6.

3.2 Determination of Precedence Relationships.

During this phase, the statements stored in the associative memory are analyzed. As a result, it generates two additional parts of the associative memory, namely, the dictionary and the precedence matrix.

The dictionary is similar to the directory of the associative memory, but contains more information about each data name used in the specification. Also, for a data name there can exist only one directory entry, but the same name can be represented by more than one dictionary entries. These different dictionary entries identify the different representations of the same name, which have different parents or different subscripts.

The precednece matrix contains precedence relationships between the different dictionary entries. These precedence relationships are determined by analyzing the description of each statement. This precedence matrix is also called as weighted adjacency matrix which represents a directed graph defining the precedence relationships (edges of the graph) between the different dictionary entries (nodes of the graph).

The directed graph represented by the precedence matrix is further analyzed during the third phase of the MODEL processor (see also the section 3.3).

During the generation of the dictionary and the precedence matrix, some forms of incompleteness are resolved by generating additional data description statements. This partial resolution of incompleteness simplifies the analysis of the graph later on. This phase is further described in Chapter 8.

3.3 Analysis of the Directed Graph.

In this phase, the directed graph is analyzed for inconsistency, incompleteness and cycles. Inconsistency can be due to invalid subscript range specification, or due to inconsistent use of subscript names. Incompleteness can be due to the omission of the data description statements for some data names or the omission of an assertion that evaluates a field of an output file. Most of these incompletenesses and inconsistencies are automatically resolved and are reported to the user for his verification. Unresolvable errors are also reported to the user, and the user response is appropriately processed.

The directed graph should not contain any cycles. If it does, it cannot be sequenced. Therefore, all the assertions that form a cycle are further analyzed; and if they represent a set of simultaneous equations, then they are "triangularized" using standard solution methods. If block triangularization is not possible, then the error is reported to the user. This phase is further described in Chapter 9.

3.4 Documentation.

This phase generates some reports from the analysis of the MODEL specification. One of the generated reports is the cross reference report which provides an alphabetical listing of all the names provided by the user. For each name, the report provides the statement number in which the name was described, the statement numbers of statements in which it was referenced, and the attributes of the name.

The second report is the formatted MODEL specification, which is a complete MODEL specification of the problem. This formatted specification is also saved in the MODEL statement library, so that any section of it can be referenced by another user, by appropriately identifying the module name and the section name. This phase is further described in Chapter 10.

3.5 Sequencing.

In this phase, the directed graph is sequenced using standard topological sorting techniques. As a result, a flow chart-like report is generated, which describes the sequence and control logic of the desired module. This phase of the MODEL processor has not been implemented; however, the techniques used in the earlier version of the system (PRY76A) can be used in the design of this phase (see also the Chapter 11).

3.6 Code Generation.

In this phase, the program for the module is generated in a high level language, PL/1. It uses the flowchart output of the sequencing phase and inserts appropriate I/O commands and control statements, thus obtaining an executable PL/1 program. This phase of the MODEL processor has not been implemented; however, the techniques used in the earlier version of the system (PRY76A) can be used in the design of this phase (see also the Chapter 11).

3.7 Compilation and Execution of the Generated Program.

The program generated in the previous phase is now compiled using the PL/1 Optimizing Compiler, thus generating an object module or load module. This object or load module can then be executed by traditional means.

CHAPTER 4

USER REFERENCE MANUAL FOR MODEL

This chapter serves as a reference manual for using the MODEL language. It describes in detail, the various statements in MODEL. For each statement, its purpose, syntax, and semantics are given, along with some examples.

As an illustrative example, a transaction processing problem is described in MODEL in section 4.8.

4.1 General Information.

A specification of a module in MODEL consists of a set of statements which describe the desired module. These statements can occur in any order. However, the different types of statements can be divided into the following three sections.

(1). Header section: the statements in this section describe the source and target files used in the module, and any reference to other sections of the MODEL statement database.

(2). Data Description section: the statements in this section describe the media, files, records, groups, fields, interim and subscript names.

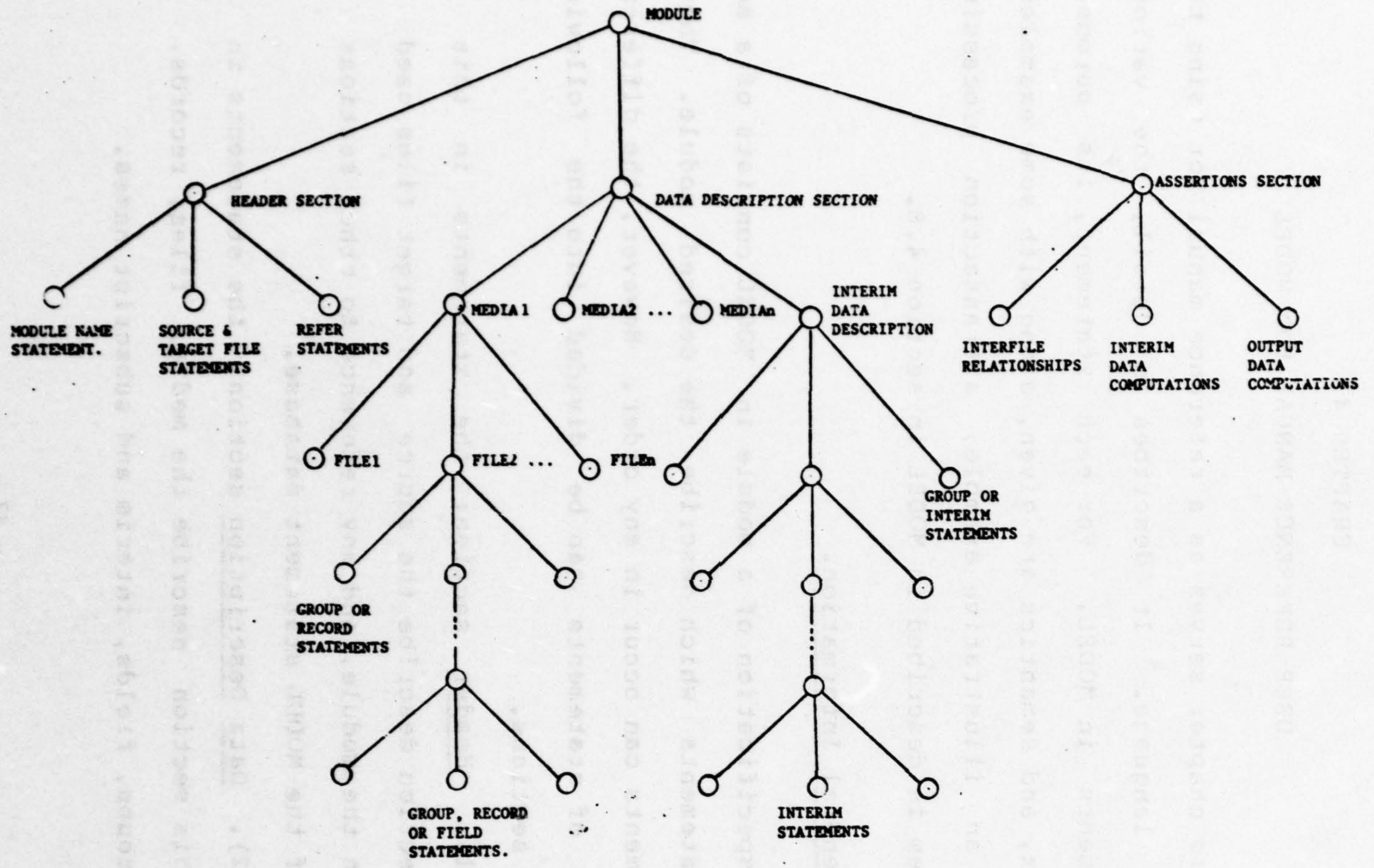


Figure 4.1. Organization of MODEL statements in the form of a TREE Structure.

(3). Assertions section: The statements in this section describe the relationship between the different data names used in the specifications.

The organization of different MODEL statements is shown in the form of a tree structure in Figure 4.1.

4.1.1 Syntax Notation.

In the description of the syntax of MODEL statements below, upper case letters refer to specific MODEL vocabulary or to user provided names and angle-bracketed < > lower case letters refer to a generic class for which a specific item needs to be substituted. The symbol '::=' means "is defined as". Square brackets [ ] indicate optional portions of the statements. Asterisks following square brackets [ ]* signify repetition zero or more times. The "|" symbol means "or", and is used for alternatives.

The following convention will be used in the description of syntax in the following sub-sections: If a syntax statement exceeds a line, then, it is split into two or more lines, the second and subsequent lines being indented by at least 4 blanks. Also, if a keyword in a statement has many representations, and some of those representations are equivalent, then only those representations that are equivalent are indicated on the same line, the others are indicated on subsequent lines, properly indented (for example, see the syntax of <org-type> in section 4.4). However, if none of the representations are equivalent, then all the representations may appear in the same line (for example, see the syntax of <disp-type> in section 4.4).

The formal syntax of MODEL is provided in section 4.7 in Extended Backus Normal Form (EBNF) specification language as a supplement.

4.1.2 Format of MODEL Statements.

MODEL statements must be in one of the following formats:

(1). Fixed (or Fixed-blocked) with record size = 80.

(2). Variable (or variable-blocked) with maximum record size = 80.

Any MODEL statement can span the first 72 characters of each record. Columns 73 thru 80 specify the line number associated with the statement. The line numbers in consecutive records need not be in sequence. If no line number is specified (that is, if columns 73 thru 80 contain non-numeric data), then the MODEL processor assigns a line number to each statement, which is the same as the record number. A record can contain any number of statements, or a statement can span more than one record. In the latter case, those records that contain the same statement must be in sequence.

Note that any data in a record beyond column 73 is ignored.

4.1.3 Names.

Whenever a <name> is indicated in the statements of MODEL, it may consist of 1 to 31 characters described below. Strictly speaking, only the names ABSENT, ENDGRP, ENDFILE, EXIST, LENGTH, POINTER and SUBSET are the reserved words in MODEL. However, to avoid ambiguity, the following key words also may not be used for names: MODULE, SOURCE, TARGET, END, SECTION, REFER, HELP, IF. The special reserved words mentioned above are described in section 4.7.

4.1.4 Character Set.

The characters which can be used to form names can be any combination of 1 to 31 letters, digits, or special characters defined below, but the first character of a name must be a letter.

<letters>::= A | B | ... | Y | Z | # | $ | @

<digits> ::= 0 | 1 | ... | 9

<special-character>::= _

4.1.5 Integers.

Whenever an <integer> is indicated it may be any combination of digits with the value from 0 to 32767, except where further limits are indicated.

4.1.6 Indication of End of Statement.

The character ';' can be used optionally as a delimiter to indicate end of a statement. As this delimiter is optional, it will not be shown in the syntax of statements that follow.

There is one risk of not using ';' at the end of each statement: that is, if there is any syntactical error in a statement, the record containing the statement is discarded. Therefore, if two statements were specified in a record, and a syntactical error was found in the first statement of that record, then the second statement is lost if no ';' was used to separate the two statements in that record. Therefore it is recommended to use ';' at the end of each statement, or specify only one MODEL statement on each record(or line). This is further described in chapter 10.

4.1.7 Qualified Names.

Whenever a <qualified-name> is indicated in a statement, it may consist of 1 to 10 <name>s connected together as shown in the following syntax:

```
<qualified-name>::= <name> [.<name>]*
```

4.1.8 Some Common Terms.

In the following, the syntax of some commonly used terms are given:

```
<is>::= IS | ARE | =
<qualified-names>::= <qualified-name> [,<qualified-name>]*
<list-of-names>::= <name> [,<name>]*
<pname>::= [<name>.]*
<dd-name>::= <pname> <name>
<list-of-dd-names>::= <dd-name> [,<dd-name>]*
```

4.1.9 Error Handling.

If a syntactical error is detected in a MODEL statement, then that error is reported to the user by:

(1). Displaying the input line which contains the MODEL statement in error,

(2). Indicating the position of error by a '|' sign, and

(3). Displaying the error code.

For example, in the following statement:

```
A    B(1) + C(X);
```

a '=' sign is expected between the letters A and B. The Processor will display the error as shown below:

```
A    B(1) + C(X);
     |
WEQUAL ERROR. INVALID TEXT BEGINNING 'B' IN LINE XXX.
        STATEMENT TILL THE NEXT SEMICOLON IS DISCARDED.
```

The error code in this case is WEQUAL, and if the description of the error code is required, then the user might type:

```
HELP WEQUAL
```

(see also the description of the HELP statement in the next chapter).

Also, whenever an error is detected, the input statement till the next semicolon is discarded. However, if no semicolon is found in the line, then the whole line is discarded.

4.2 A Mini-manual for MODEL.

This section gives a simplified description of the MODEL statements using some examples. The complete syntax and the detailed description of each statement is given in the next three sections.

Essentially, there are two types of statements in

MODEL:

(1). Data description statements which describe the structure of a file and its sub-components, and

(2). Assertions that define the relationship between different data items.

4.2.1 Data Description Statements.

The data description statements can be described by the following simplified syntax:

```
<ddstmt>::= <name> IS <stmt-type> [ ( <arguments>)]
<stmt-type>::= FILE | RECORD | GROUP | FIELD
                | INTERIM | SUBSCRIPT
```

The above statements will be described using the example in Figure 4.2. The statements in this figure describe the structure of a file called INPUT. It consists of a set of records (number of records in the set is indicated by the repetition '*', which means that it is unknown and is to be determined from the end-of-file condition) denoted by INREC (statement 2). Each record consists of a field "NO_OF_ITEMS", and a group of two fields denoted by ITEMG. The number of ITEMG in each record INREC is determined by the field NO_OF_ITEMS. ITEMG consists of two fields denoted by ITEM# and QUANT (statements 5 and 6). Both of these items are 3 characters long. Therefore, the length of each ITEMG is 6, and the length of each record is given by the expression:

```
    2 + NO_OF_ITEMS * 6
```

Figure 4.3 gives the physical organization of the INPUT file, in which the first two records are shown. In this figure, the first record is shown to have two ITEMG and the second, only one. The offset of individual item (from the beginning of the corresponding record) is shown at the top of the figure.

In a data description statement, the first argument (for example, "INPUT" in statement 2 of Figure 4.2) specifies the parent name of that data item. This linking between the data name and its parent allows us to interpret every file description in the form of a tree structure. For example, the tree structure for the statements in Figure 4.2 is shown in Figure 4.4. The other arguments of a dd statement describe the repetition of that item (as in statements 2 and 4 of Figure 4.2) or the length and attributes of that item (as in statements 3, 5 and 6 of Figure 4.2).

```
INPUT IS FILE;
  INREC IS RECORD(INPUT,(*));
    NO_OF_ITEMS IS FIELD(INREC,PIC'99');
    ITEMG IS GROUP(INREC,(NO_OF_ITEMS));
      ITEM# IS FIELD(ITEMS,PIC'999');
      COUNT IS FIELD(ITEMS,PIC'999');
```

Figure 4.2. A Sample File Description.

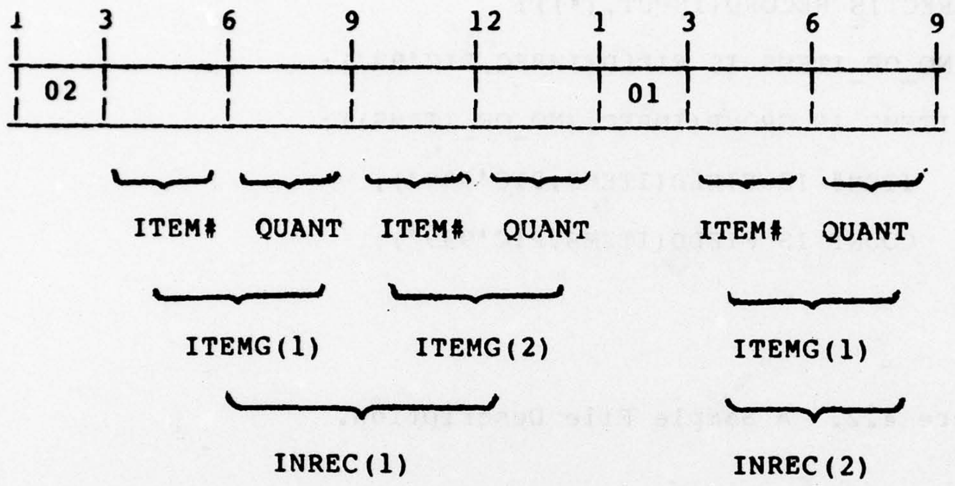


Figure 4.3. Layout of the File Described in Figure 4.2.

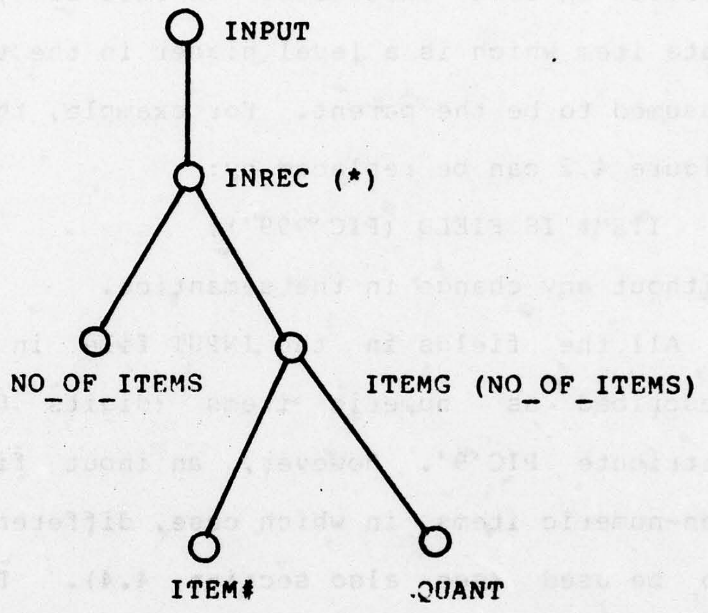


Figure 4.4. Tree Structure of the INPUT file Described in Figure 4.2.

The first argument (that is, parent name) can be omitted in some instances. In this case, the preceeding data item which is a level higher in the tree structure is assumed to be the parent. For example, the statement 5 in Figure 4.2 can be replaced by:

```
ITEM# IS FIELD (PIC'999');
```

without any change in the semantics.

All the fields in the INPUT file in Figure 4.2 were described as numeric items (digits 0-9) using the attribute PIC'9'. However, an input field may contain non-numeric items, in which case, different attributes has to be used (see also section 4.4). For example, the statement:

```
NAME IS FIELD(CHAR(20))
```

describes a field which is a string of length 20.

The above examples give a simplified description of some of the data description statements in MODEL. The detailed description of each data description statement is given in section 4.4.

4.2.2 Assertions.

A simplified syntax for the assertions is given below:

```
<assertions>::= <for-clause> <assertions>
              | <boolean-clause> <assertions>
              | <simple-assertion>

<simple-assertion>::= <data-name> = <expression>
```

In the above syntax, the <for-clause> specifies a set of ranges for the subscript names used in the assertion; and the <boolean-clause> is a boolean expression used to conditionally execute the following assertion. The <simple-assertion> defines the relationship between the data-item on the LHS of the equal sign and the data items on the RHS. RHS can be any expression using regular operators like, +, -. *, /, >= ..., etc. The data items on the LHS or RHS can be optionally subscripted. Functions can also be used in the expressions on the RHS. Some examples of assertions are shown in Figures 4.5 and 4.6.

```
ITEM_SUM = QUANT * PRICE(ITEM#);
TOTAL = SUM(ITEM_SUM);
```

Figure 4.5. Some examples of Assertions.

```
FOR I = 2 TO NO_OF_ITEMS
        ITEM_SUM = QUANT(I) * PRICE(ITEM#(I)) * 0.9;
TOTAL = QUANT(1) * PRICE(ITEM#(1)) +
        SUM(ITEM_SUM);
```

Figure 4.6. Some Additional Examples of Assertions.

Assuming that the data name PRICE is an array containing the price of each item, and the data names QUANT and ITEM# refer to the fields described in Figure 4.2, then the two assertions in Figure 4.5 compute the total price (in TOTAL) of all items in the record INREC. The assertions in Figure 4.6 also perform the same function; however, a rebate of 10% is applied to all items except the first.

The data name ITEM_SUM used in Figures 4.5 and 4.6 is assumed to be a temporary variable (INTERIM) and no data description statement for that name is necessary. However, for completeness sake, that name can be described just like a field statement (see also Figure 4.2).

The data name TOTAL used in Figures 4.5 and 4.6 is assumed to be a field in the output file (SYSPRINT). Again, a dd statement for this name is not needed, however, for the sake of completeness, a set of statements can be supplied as in Figure 4.2.

The first statement in Figure 4.6 uses a "for-clause" which defines the range of the subscript (I, in this case) used in the relation that follows it. If no range is supplied, then the entire range of the subscript is used as the range. For example, the first assertion in Figure 4.5 is equivalent to:

```
FOR I = 1 TO EXIST.INREC & J = 1 TO NO_OF_ITEMS
   ITEM_SUM(I,J) = QUANT(I,J) *
         PRICE(ITEM#(I,J));
```

Fortunately, the system propagates the appropriate subscript names to each data name, and therefore the simplefied form of the data names can be used as shown in Figures 4.5 and 4.6.

The EXIST.INREC used in the above assertion defines the size of the array INREC which is computed from the end-of-file delimitor. This is described further in later chapters.

The above examples give a simplified description of assertions in MODEL. The detailed description of assertions and that of other related statements is given in section 4.5.

4.3 Header Section.

The header of MODEL specification consists of four statements, module name statement, source file statement, target file statement and refer statement, described below.

4.3.1 Module Name Statement.

<u>Purpose</u>: to give a name to the desired module.

Syntax:

```
<module-name-statement>::= MODULE: <name>
        |   <name> [ IS ] MODULE
```

Semantics: <name> is used as the module name. If no module statement exists, then the name '$MODULE' is given to the module.

Example:

```
     MODULE: DEPSALE
```

4.3.2 Source File Statement.

Purpose: to indicate the names of those files which are source or input to the desired module.

Syntax:

```
<source-file-statement>::= <keyword1> [ <keyword2> ] :
                                        <list-of-qnames>
        | <list-of-qnames> <is> <keyword1> [ <keyword2> ]
```

where:

```
<keyword1>::= SOURCE | SOU

<keyword2>::= FILES | FILE

<list-of-qnames>::= <qnames> [, <qnames>]*

<qname>::= <name> [ . <name> ]
```

Semantics: the qualified names, given by <list-of-qnames> are source files to the module. If any <qname> in list contains two <name>s, then the first <name> is the module name in which the source file is described and the second

<name> is the file name itself.

If no file description statement for the specified file exists, then the corresponding file description is obtained from the MODEL statement database.

Example:

SOURCE FILES: DEPSALE.MASTER, TRANS

The above statement indicates that the files named MASTER and TRANS are source files to the module, and that the file description of MASTER is to be obtained from the description of the module DEPSALE in the MODEL statement database.

4.3.3 Target File Statement.

Purpose: to indicate the names of those files which are targets or outputs of the desired module.

Syntax:

```
<target-file-statement>::= <keyword3> [<keyword2>] :
                                    <list-of-qnames>
        | <list-of-qnames> <is> <keyword3> [<keyword2>]
```

where:

<keyword3>::= TARGET | TAR

and the items <keyword2> and <list-of-qnames> has the same syntax as in <source-file-statement> described in section 4.3.2.

Semantics: The qualified names given by <list-of-qnames>

are target files to the module.

Example:

TARGET FILE: MASTER

Note that a file can be both source and target, such as a file to be updated.

If the source and/or target file statements were not specified, then the MODEL processor automatically resolves the files used in the module as source or target, if any of the fields in that file is used as source or target respectively.

4.3.4 Refer Statement.

Purpose: Adds the specified sections from the MODEL statement data base to the MODEL specification for the module.

Syntax:

```
<refer-statement>::= <keyword>: <qualified-names>
where:
<keyword>::= REFER | REF
```

Semantics: The qualified names given by <qualified-names> are the section names to be included in the MODEL specification of the module. The first name in each qualified name is the module name in which the section was specified.

If the qualified name contains only one name, then the name is assumed to be the module name and all the MODEL statements in the specified module are included. Example:

```
REFER: DEPSALE.TRANS.GRP
```

This statement includes the subsection GRP in the section TRANS of the module DEPSALE. TRANS can be any data name, and GRP is one of its descendants.

Note that any branch of the tree structure representing a MODEL specification can be referenced be referring the node in the tree in which the branch starts. For example, for the tree structure of a MODEL specification, shown in Figure 4.7, the statement:

```
REFER: DEPSALE.DATA_DESCRIPTION.DISK.TRANS
```

or the statement:

```
REFER: DEPSALE.TRANS;
```

can be used to refer the data description statement of the file TRANS, and the data description statements of all its descendents.

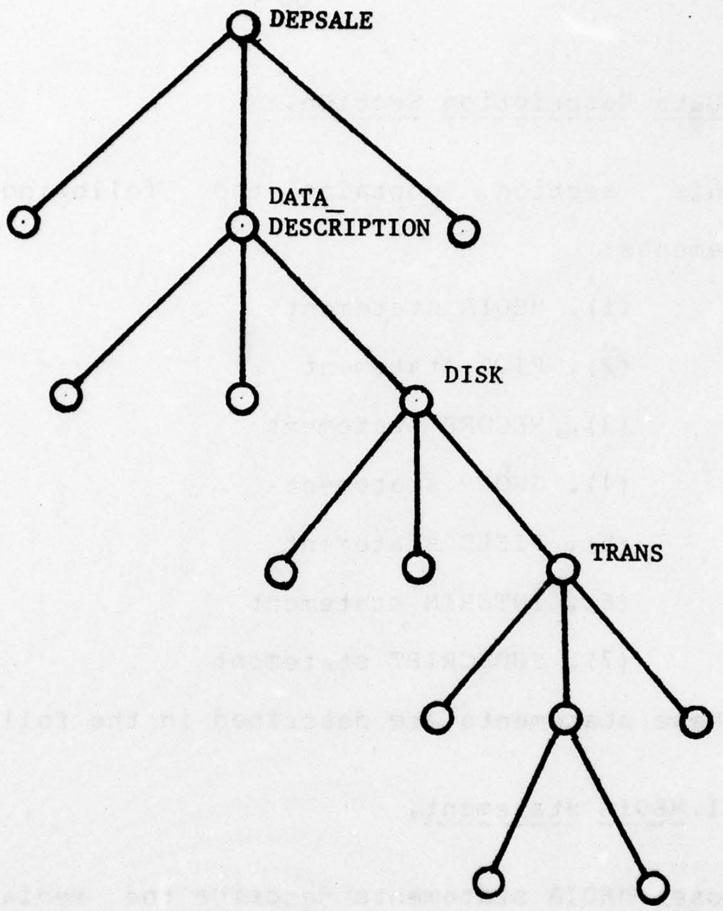


Figure 4.7.

4.4 Data Desciiption Section.

This section contains the following tyoes of statements:

(1). MEDIA statement

(2). FILE statement

(3). RECORD statement

(4). GROUP statement

(5). FIELD statement

(6). INTERIM statement

(7). SUBSCRIPT statement

These statements are desciibed in the following.

4.4.1 MEDIA Statement.

Purpose: MEDIA statements desciibe the media in which the source oi target files ieside.

Syntax:

```
<media-statement>::= <list-of-names> [<is>] <keywoidl>
            [(<aiqument> [,<aiqument>]*)]
```

where:

```
<keywordl>::= MEDIA | MED

<aiqument>::= <common-aiguments>
            | <tape-arquments>
            | <teiminal-aiquments>

<common-aiquments>::= <blocksize-spec>
            | <iecordsize-spec>
            | <oiqanization-spec>
            | <iecfm-spec>
```

```
                | <unit-spec>
                | <disp-spec>

<tape-arguments>::= <tape-label-spec>
                | <parity-spec>
                | <charcode-spec>
                | <trks-spec>
                | <density-spec>
                | <int-label-spec>
                | <ext-label-spec>

<terminal-arguments>::= <tab-spec>
                | <page-spec>
                | <line-spec>

<blocksize-spec>::= [<keyword2> [<is>]] <integer>

<keyword2>::= BLOCKSIZE | BLKSIZE | BS

<recordsize-spec>::= [<keyword3> [<is>]] <integer>

<keyword3>::= RL | RECORDSIZE | RECSIZE | LRECL

<organization-spec>::= [<keyword4> [<is>]] <org-type>

<keyword4>::= ORG | ORGANIZATION

<org-type>::= SAM | SEQ | SEQUENTIAL
                | ISAM | INDEXED SEQUENTIAL | INDEXED | IS
                | DIRECT | REGIONAL | REG | DIR | REG1
                | REG2
                | REG3

<recfm-spec>::= [<keyword5> [<is>]] <recfm-type>

<keyword5>::= RECFM | RECORD_FORMAT | RF

<recfm-type>::= FIXED | F
                 | FIXED BLOCKED | FB
                 | VARIABLE | V
                 | VB | VAR BLOCKED
                 | VAR SPANNED | VS
                 | UNDEFINED | U

<unit-spec>::= [<keyword6> [<is>]] <unit-type>

<keyword6> ::= UNIT | DEVICE
```

```
<unit-type>::= PUNCH
             | CARD
             | TAPE | 3400-4
             | 3400-1
             | DISK | 3330
             | 3330-1
             | 2314
             | 2311
             | 2305
             | TERMINAL | SYSOUT | PRINT | SYSPRINT |SYSIN
             | OLS

<disp-spec>::= [<keyword7> [<is>]] <disp-type>

<keyword7>::= DISP | DISPOSITION

<disp-type>::= OLD | SHR | NEW | MOD

<tape-label-spec>::= <keyword8> [<is>] <label-type>

<keyword8>::= TLABEL | TAPE_LABEL | TL

<label-type>::= SL | IBM_STD | ANSI_STD
              | NL | NONE
              | BLP | BYPASS

<parity-spec>::= <keyword9> [<is>] <parity-type>

<keyword9>::= PARITY | PAR

<parity-type>::= ODD | EVEN

<charcode-spec>::= <keyword10> [<is>] <cc-type>

<keyword10>::= CHAR_CODE | CC | CODE

<cc-type>::= EBCDIC | BCD | ASCII

<page-spec>::= <keyword11> [<is>] <integer>

<keyword11>::= PAGESIZE | P | LINES_PER_PAGE

<line-spce>::= <keyword12> [<is>] <integer>

<keyword12>::= LINESIZE | WIDTH | PAGE_WIDTH | LINE_WIDTH

<trks-spec>::= <keyword13> [<is>] <trk-no>
```

```
<keyword13>::= TRKS | NO_TRKS | TRACKS
<trk-no>::= 7 | 9
<density-spec>::= <keyword14> [<is>] <density-type>
<keyword14>::= DEN | DENSITY
<density-type>::= 200 | 556 | 800 | 1600
<int-label-spec>::= <keyword15> [<is>] <name> [,<name>]*
<keyword15>::= IL | INT_NAME | ILABEL
<ext-label-spec>::= <keyword16> [<is>] <name> [,<name>]*
<keyword16>::= EL | VOLSER |VOLUME | EXT_NAME | ELABEL
<tab-spec>::= <keyword17> [<is>] <tab-parms>
<keyword17>::= TAB
<tab-parms>::= (<integer> [,<integer>]*)
```

Semantics: The list of names given by <list-of-names> specifies the names of the storage medium, in which the input/output files reside. The arguments specify the attributes of the media. Though, some of the arguments are associated with a particular file in the media, the attributes that are common to all files in the media can be specified in the media statement itself. In the following, a short description of each argument is given.

<blocksize-spec>: specifies the block size which is equal to the sum of:

(1). the lengths of all records in the block

(2). any control bytes required.

The relationship of the block size to the record length depends on the record format. For FB format, the blocksize must be a multiple of record length; for VB format, the block size must be equal to or greater than 4 + the lengths of all records in the block. The maximum value of the blocksize is 32760. If this argument is not specified, then the blocksize is assumed to be the value given by the following expression:

recordsize * max(1,6400/recordsize)

<recordsize-spec>: specifies the record length. This is the sum of:

(1). the length required for data. For variable length and undefined length records, this is the maximum record length;

(2). 4 in the case of variable length records, 0 otherwise.

The maximum record length is 32756 in the case of variable length records, 32760, otherwise. If this argument is not specified, then the record length is assumed to be the computed length of record.

<organization-spec>: specifies the record organization in the file. Three types of record organization are allowed:

(1) Sequential

(2) Indexed sequential (or indexed)

(3) Regional (or direct)

There are three sub-types in the Regional type of organization. These sub types are indicated as REG1, REG2 and REG3 (corresponding to the three PL/1 types: Regional(1), Regional(2) and Regional(3) respectively). Note that the types (2) and (3) are allowed only for disk files. If this argument is not specified, then the organization is assumed to be SAM.

<recfm-spec>: specifies the record format of the file. If this argument is not specified, then the record format is assumed to be FIXED, if blocksize = recordsize, FB, if Blocksize = n*Recordsize (where n is an integer) UNDEFINED, otherwise.

<unit-spec>: specifies the device name of the media. If this argument is not specified, then UNIT = OLS is assumed.

<disp-spec>: specifies the disposition of the data set. If this argument is not specified, then DISP = OLD is assumed.

<tape-label-spec>: specifies the type of label of a tape. If standard label tape, then the internal label name must be specified (see the description of <int-label-spec> below). BYPASS (or BLP) means that tape

label processing is to be bypassed; NONE (or NL) denotes no tape label.

If this argument is not specified, then TL = SL is assumed.

<parity-spec>, <trks-spec> and <density-spec> specify parity, number of tracks and density of the tape respectively. If any one of these arguments is not specified, then correspondingly, PARITY = ODD, TRKS = 9, or DENSITY = 1600 is assumed.

<int-label-spec>: specifies the internal label(s) of the tape(s) if they are standard labelled. Each label name (<name> in <int-label-spec>) should be 6 characters long. The number of <name>s must be equal to the number of tapes that the file uses. If this argument is not specified, then the internal label is assumed to be the same as the external label.

<ext-label-spec>: specifies the external label(s) of the tape(s). Each label name should be 6 characters long. The number of <name>s must be equal to the number of tapes that the file uses. If this argument is not specified, then EL = XXXXXX is assumed.

<charcode-spec>: specifies the character code of the data. If this argument is not specified, then CODE = EBCDIC is assumed.

<page-spec>: specifies the page size (that is, number of lines per page) for the print files. If the specified size is > 66 then a warning message is reported. If this argument is not specified, then PAGE_SIZE = 60 is assumed.

<line-spec>: specifies the maximum length of a line for the print files. If the specified length is > 133, then a warning message is reported. If this argument is not specified, then line size is assumed to be the same as the record length.

<tab-spec>: specifies the tab positions for the print files. At most 10 tabs can be specified and the values must be in ascending order, the maximum being not greater than 133.

Example1:

```
 INVDISK IS MEDIA(ISAM,DISK,FIXED,180)
```

The above statement is the same as the following statement:

```
 INVDISK IS MEDIA(ORG=ISAM, UNIT=DISK, RECFM = FIXED,
           BLOCKSIZE = 180)
```

Example2:

```
 TRANS IS MEDIA(TAPE, 3200, 80,EL=X00010)
```

The above statement specifies a standard label tape (with external label = X00010), the descendant files of which have blocksize of 3200 and recordsize of 80.

4.4.2 FILE Statement.

Purpose: To describe a file and some of its attributes.

Syntax:

```
<file-statement>::= <list-of-gnames> [<is>]
        <keyword1> [(<parent-name> [,<argument>]*)]
```

where:

<list-of-gnames> has the same syntax as the corresponding syntax in <source-file-statement> and,

```
<keyword1>::= FILE | REPORT | FIL | REP

<argument>::= <common-arguments>
            | <tape-arguments>
            | <disk-arguments>
            | <terminal-arguments>

<common-arguments>::= <blocksize-spec>
                    | <recordsize-spec>
                    | <organization-spec>
                    | <recfm-spec>
                    | <unit-spec>
                    | <disp-spec>
                    | <key-spec>
                    | <dsn-spec>

<tape-arguments>::= <tape-label-spec>
                  | <parity-spec>
                  | <charcode-spec>
                  | <trks-spec>
                  | <density-spec>
                  | <int-label-spec>
                  | <ext-label-spec>
                  | <startfile-spec>

<terminal-arguments>::= <tab-spec>
                      | <page-spec>
                      | <line-spec>

<disk-arguments>::= <space-spec>
```

The arguments <startfile-spec>, <key-spec>, <dsn-spec> and <space-spec> have the following syntax:

```
<start-file-spec>::= <keyword2> [<is>] <integer>
<keyword2>::= FILE# | START_FILE
<key-spec>::= KEY [<is>] <qualified-name>
<dsn-spec>::= <keyword3> [<is>] <dsn-itself>
<keyword3>::= DSN | DSNAME | DATASET | FILE_NAME
<dsn-itself>::= <dsn-name>  [(<member>)]
<member> ::= <name> | [<sign>] <integer>
<sign>::= + | -
<space-spec>::= <keyword4> [<is>] <space-pars>
<keyword4>::= SPACE
<space-pars>::= ( <units> [,[<integer>]
          [,[<integer>] [,<integer>]]] [[,]
          <rlse>] )
<units>::= TRACKS | TRACK | TRK
         | CYL | CYLS
         | BLOCKS | BLK
         | <integer>
<rlse>::= R | RLSE | RELEASE | REL
```

The other terms in <tape-arguments>, <common-arguments> and <terminal-arguments> have the same syntax as the corresponding arguments in the media-statement described in section 4.4.

Semantics: The list of names given by <list-of-names>

specifies the names of the input/output files used in the specification. Each <qname> in the <list-of-qnames> can contain oneor two names. In the former case, the only name in <qname> is the file name; and in the latter case, the first name in <qname> is the name of the media (which is the parent of file) and the second name is the file name itself. If <qname> contains two names, then <parent-name> shold not be specified (see the syntax of file statement) in the argument; otherwise, the parent name specifies the nameof the parent media. The file names are analogous to the file names in PL/1, and should not be confused with the data set name. The data set name that the file represents is given by the argument <dsn-spec>, which is described below. The arguments that are common for media and file statements have the semantics described in section 4.4. The semantics of the arguments <startfile-spec>, <key-spec>, <dsn-spec> and <space-spec> are described briefly in the following:

<u><startfile-spec></u>: specifies the file number in the tape in which the file resides. If this argument is not specified, then FILE# = 1 is assumed.

<u>&lt;key-spec&gt;</u>: specifies the key name (which is given by <qualified-name> in <key-spec>) associated with the file. If the file name is sequential file, then the key name

must be one of the fields in the file; otherwise, the key name can be any interim or source data name used in specifying the module. If this argument is not specified, and the file is an indexed file, then the first field name in the file is assumed to be the key name.

<dsn-spec>: specifies the data set name (given by <dsn-name>). The dsn-name consists of names delimited by the character '.', the syntax of which is given below:

```
    <dsn-name>::= <name>[.<name>]*
```

Each <name> in <dsn-name> can be at most 8 characters long, and the total length of <dsn-name>, including delimiters is 44. The data set name can also contain a member name (given by <name> in <member>) if the data set is a partitioned data set, or a generation number (given by <integer> in <member>) in the case of a generation data set. The member name or the generation number must be enclosed in parenthesis as shown in the syntax of <dsn-itself>. The length of member name is also limited to 8 characters. The following gives a list of valid data set names:

```
        ABC.TRANS
        ABC.TRANS(JULY)
        ABC.TRANS(-1)
```

If this argument is not specified, then the file name itself is used as the data set name.

<space-spec>: specifies the amount of space to be allocated. This argument is allowed only if the file is an output file with disposition (DISP) = NEW. The <units> in the <space-pars> specifies the basic unit of space allocation, which can be tracks, cylinders, or an integer which indicates the number of bytes. If BLK (or BLOCKS) is specified, then block size is used for the <units>. The first <integer> in <space-pars> indicates the number of units of primary allocation; whereas the second and third integers indicate secondary allocation amount and the number of units to be allocated for the directory (in the case of partitioned data set). The last term <rlse> indicates to release the unused space inthe data set at the end of job.

If this argument is omitted, then SPACE=(TRK,10,1,1) is assumed.

Example:

```
INVDISK.INVEN IS FILE(DSN=OLD.INVEN,KEY=STOCK#)
```

This statement specifies the file name INVEN (the parent of which is the media, INVDISK), which represents the data set name OLD.INVEN and the key name of the file is the field STOCK#.

Note that if some attributes of a file is specified in both media and file statements, then the corresponding attribute in the file statement is used. For example, the following two statements:

```
    DISK1 IS MEDIA(SAM,BLOCKSIZE=6400)
        FILE1 IS FILE(DISK1,BLOCKSIZE=3200)
```

are equivalent to the following two statements:

```
    DISK1 IS MEDIA(SAM)
        FILE1 IS FILE(DISK1,BLOCKSIZE=3200)
```

4.4.3 RECORD Statement.

Purpose: to describe the record of a file and its attributes.

Syntax:

```
<record-statement>::= <list-of-dd-names> [<is>]
          <keyword1> [(<parent-name> [,<argument>]*)]
```

where:

```
<keyword1>::= RECORD | REC
<argument>::= <occ-desc> | (<occ-desc>) | <key-spec>
<occ-desc>::= <occ-list> [,<occ-list>]*
<occ-list>::= [<occ>:]<occ>
<occ>::= * | <integer> | <name>
```

Semantics: The name in each <dd-name> in the <list-of-dd-names> specifies a record name, the attributes of which are given by the rest of the statement. The parent name of the record can be specified either by prefixing the parent name in <dd-name> itself, or by the first argument, <parent-name>.

The <occ-desc> specifies the size and dimension of the record. The dimension of the record is given by the number of <occ-list> in <occ-desc>. The first <occ> in each <occ-list> indicates the lower bound of the corresponding dimension, and the second <occ> indicates the upper bound of the corresponding dimension. If the first <occ> is omitted, then 1 is assumed. If '*' is used to specify the number of repetitions, then the number of repetitions of the record is determined by the end-of-file delimiter. If the argument <occ-desc> was not specified at all, then the number of repetitions is assumed to be '*'.

The argument <key-spec> specifies the key-name, if any, associated with the record. The semantics of this argument is the same as the corresponding argument in <file-statement>.

Example1:

```
    REC1 IS RECORD (INVEN,(*))
```

This statement describes the record, REC1, which is a descendant of a data name INVEN. The number of repetitions of the record is to be determined by the end-of-file delimiter or by setting of the "ENDGRP" variable ENDGRP.REC1 in some assertion.

Example2:

```
    INVEN.REC2 IS RECORD(10);
```

This record statement describes the same record name as in example 1, however, in this case, the number of repetitions of the record is fixed and is equal to 10.

4.4.4 GROUP Statement.

Purpose: To describe a group in a file structure (or in a record structure) and all its attributes.

Syntax:

```
<group-statement>::= <list-of-dd-names> [<is>]
         <keyword1> [(<parent-name> [,<argument>]*)]
 where:
<keyword1>::= GROUP | GRP | GRO
```

The item <argument> has the same syntax as the corresponding item in RECORD statement.

Semantics: The name in each <dd-name> in the <list-of-dd-names> specifies a group name, the attributes of which are given by the rest of the statement. The <argument> has the semantics as the corresponding <argument> in RECORD statement.

Examplel:

    GROUPl IS GROUP(INVEN.RECl,(10,20))

This statement describes the attributes of a group, GROUPl, the parent of which is given by the qualified name INVEN.RECl. The dimension of the group is 2 and the size of the first and the second dimensions are 10 and 20 respectively.

4.4.5 FIELD Statement.

Purpose: To describe a field in a group or a record structure and all the attributes of the field.

Syntax:

```
<field-statement>::= <list-of-dd-names> [<is>]
        <keywordl> [(<parent-name> [,<argument>]*)]
```

where:

```
<keywordl>::= FIELD | FIE | FLD

<argument>::= <occ-desc> | <field-desc>

<field-desc>::= <char> [(<length>)] [<var>]
          | BIT [(<length>)] [<var>]
          | <bin> [<fixed-or-float>] [<precision>]
          | <dec> [<fixed-or-float>] [<precision>]
          | [<dec>] <fixed-or-float> [<precision>]
          | <pic> ['<picture>']  | <num> [<length>]

<char>::= CHA | CHAR | CHARACTER

<length>::= * | <integer> | <qualified-name>
```

```
<var>::= VAR | VARYING

<bin>::= BIN | BINARY

<fixed-or-float>::= FIXED | FIX
                  | FLOAT | FLT | FLO

<precision>::= (<integer>[,<integer>])

<dec>::= DEC | DECIMAL

<pic>::= PIC | PICTURE

<num>::= NUM | NUMERIC
```

The item <occ-desc> has the same syntax as the corresponding item in RECORD statement.

Semantics: The name in each <dd-name> in the <list-of-dd-names> specifies a field name, the attributes of which are given by the rest of the statement. The argument <occ-desc> has the same semantics as the corresponding argument in RECORD statement.

The argument <field-spec> specifies the type of field the name represents. There are essentially 7 types of fields:

(1). CHARACTER
(2). BIT
(3). BINARY [FIXED]
(4). BINARY FLOAT
(5). [DECIMAL] FIXED
(6). DECIMAL [FLOAT] or [DECIMAL] FLOAT
(7). PICTURE

The <length> associated with either CHAR type or BIT type of fields specify either (1) fixed length string, (2) dynamic length string, or (3) variable length string. A data is a fixed length string if <integer> is specified for <length>. A data is a dynamic length string if the <length> is specified by '*' or by the qualified name. A data is variable length string if the keyword VARYING is specified. A dynamic length string differs from variable length string in that, in the former case, the length of the string is fixed and is computed dynamically just before the dd-name is referenced. In the latter case, the length of string is variable, however, the buffer size allocated for the string is the maximum size specified by <integer> or by the qualified name.

The <precision> associated with either BINARY or DECIMAL data specifies the number of digits and the number of fractional digits for that data. For example, if the precision is given as (10,2) it specifies 10 digits for the data, two out of which specify the fractional digits.

The description of different field types are described further in the following.

<u>CHARACTER</u> type of fields specify character strings. Character strings includes any character recognizable by the machine. The length of character string can be fixed,

dynamic, or varying. If <length> is omitted, then the length of the character string is assumed to be one. This field type defines the same data as character string data in PL/1.

BIT type of fields specify bit strings. These represent strings of binary digits (bits). The length of bit strings can be fixed, dynamic or variable, just as in the case of character string data. This field type defines the same data as bit string data in PL/1.

BINARY FIXED field types represent 1 to 31 binary digits with an optional binary point. <precision> specifies the number of digits in each field. The default precision is (15,0). This field type defines the same data as the binary fixed point data in PL/1.

BINARY FLOAT field types specify binary floating point data. It consists of string of digits with an optional exponent. The maximum precision allowed for this type of data is 109; and the default precision is 21. This field type defines the same data as the binary floating point data in PL/1.

DECIMAL FIXED field types specify decimal fixed point data. It can represent data that contains 1 to 15 decimal digits with an optional decimal point. The default precision, assumed when no specification is made, is

(5,0). This field type defines the same data as the decimal fixed point data in PL/1.

DECIMAL FLOAT field types specify decimal floating point data. The maximum precision allowed is 33. The default precision is 6. This field type defines the same field type as the decimal floating point data in PL/1.

PICTURE field types specify numeric character data and some special character string data. The set of data that can be represented by picture field types is a superset of the set of data that can be represented in PL/1. The picture specification (which is given by <picture>) consists of a sequence of character codes enclosed in quotation marks which is part of the picture attribute. Picture character codes are used to describe the attributes of the associated data item, whether it is the value of the variable or a data item to be transmitted between the program and external storage.

Picture specifications are of two types:

(1) Character string specification and

(2) Numeric character specification.

A "character string" pictured item is one that consist of alphabetic characters, decimal digits, and blanks.

A "numeric character" pictured item is one in which the data itself can consist only of decimal digits, a decimal point, the letter E (for exponent), and optionally a plus or minus sign.

There are special characters that can be used to edit either character string data or numeric character data. These characters are briefly described in the following. Except for the picture characters, B, T, L, and P in character string data, all other picture characters in either character string data, or numeric character data have the same semantics as the corresponding picture character in PL/1. For complete description of those picture characters, the reader can refer to the PL/1 reference manual.

<u>Picture Characters for Character String Data</u>: A picture specification is assumed to be for a character string data if the picture specification consists of at least one of the characters "A" or "X", or if it contains only the characters "B", "L", "P" and "T". The syntax of character string data can be specified as follows:

```
<char-picture-specification>::= <single-item>
                [<single-item>]* [<padding>]
         | <padding>

<single-item>::= [<repetition>]<chars>

<repetition>::= (<integer>)

<chars>::= X | A | 9

<padding>::= [<repetition>] <pad-chars>
```

```
[[<repetition>] <pad-chars>]*
```

```
<pad-chars>::= B | L | P | T
```

The individual picture characters used in the picture specification is described briefly in the following:

X Specifies that the associated position can obtain any character whose internal bit configuration can be recognized by the computer in use.

A Specifies that the associated position can contain any alphabetic character or a blank character.

9 Specifies that the associated position can contain any decimal digit or a blank character.

B Specifies that a blank character is to be padded.

L Specifies that an end of line character is to be padded.

P Specifies that an end of page character is to be padded.

T Specifies that blanks are to be padded till the next tab position. This picture character is allowed only for fields in a file for which tabs were specified (see also the description of the argument <tab-spec> in MEDIA statement). If no tab was specified for the file, then this picture character is ignored.

Note that the picture specification characters B, L, P and T can be used only as a suffix of any combination of A, X and 9. These are used specially in the specification of fields in the print files, or reports. Some examples are given at the end of this sub-section.

<u>Picture Characters for Numeric Character Data</u>: The picture characters for numeric character specification may be grouped into the following categories:

(1) Digit and Decimal point specifiers (9 and V).

(2) Zero suppression characters (Z and *).

(3) Insertion characters (",", ., /, and B).

(4) Signs and Currency Symbol (S, +, - and $).

(5) Credit, Debit and Overpunch signs (CR, DB, T, I, R and Y).

(6) Exponent specifier (K and E).

(7) Scaling Factor (F).

Since the above picture characters have the same semantics as the corresponding picture character of PL/1, they will not be described further here. The description of these picture characters can be found in the reference manual for PL/1. However, some examples shown at the end of this sub-section may give an overview of some of these picture characters.

The <u>NUMERIC</u> field type is the same as the picture field type, the picture specification of which contains the character '9', the number of repetition of which is given by <length>. For example, the following field specification:

```
        PICTURE '999'
```

is equivalent to the field specification:

```
        NUMERIC (3)
```

Example1:

```
     X IS FIELD(A,CHAR(10),(20))
```

This statement defines a character string data name, X, which is a vector of 20 elements, the length of each element being ten. The data name A is the parent name of X.

Example2:

```
     X IS FIELD(A,BIT)
```

This statement defines a bit string data name, X, which is a scalar (dimension = 0) and the length of string is one.

Example3:

```
     X IS FIELD(A,BIN(15))
```

This statement defines a binary fixed point data name, X, which is a scalar and the precision of which is (15,0).

Example4:

```
     X IS FIELD (A,DEC,(10,10))
```

This statement defines a decimal fixed point data name, X, which is a matrix of size 10x10. The precision of each element is (5,0).

<u>Example5</u>:

```
    X IS FIELD (A,PIC'ZZV99')
```

This statement defines a numeric character data name, X, which is a scalar and has a length of 4. It also causes zero suppression for the first two digits in the field. As an example, if the value 1.23 is assigned to X, then the character representation of X is given by: 'b̸123', where b̸ represents a blank character. The picture character Z is the zero suppression character, and the picture character V is the decimal point specifier. In the above example, the decimal point is assumed to precede the right most two digits.

<u>Example6</u>:

```
    X IS FIELD(A,PIC'9,999')
```

This statement defines a numeric character data name, which is a scalar and causes an insertion of a comma if the arithmetic value of the field is greater than 999. As an example, if the value 1234 is assigned to X, then the character representation of X is given by: '1,234'.

4.4.6 INTERIM Statement.

Purpose: To describe the attributes of interim variable names. Interim variable names are the same as the field names, but are not contained in either source nor target files.

Syntax:

```
<interim-statement>::= <list-of-dd-names> [<is>]
            <keyword1> [(<rest-of-stmt>)]
```

where:

```
<keyword1>::= INT | INTERIM
<rest-of-stmt>::= <parent-name> [,<argument>]*
              | <argument> [,<argument>]*
```

The item <argument> has the same syntax as the corresponding item in FIELD statement.

Semantics: The arguments of interim statements have the same semantics as the corresponding arguments in the field statements. However, Interim data names may not have any parent.

Example:

```
    TEMP IS INTERIM((10,10), BIN FIXED)
```

This statement defines a binary fixed point data name, TEMP, which is a matrix of size 10x10.

4.4.7 Subscript Statement.

Purpose: To describe the attributes of a subscript name.

Syntax:

```
<subscript-statement>::= <list-of-names> [<is>]
          <keyword1> [(<rest-of-stmt>)]
```

where:

```
<keyword1>::= SUB | SUBSCRIPT

<rest-of-stmt>::= <parent-name> [,<argument>]
                | <argument>

<argument>::= <integer> [,[<integer>] [, [<integer>]
                [,<integer>]]]
```

Semantics: The names in <list-of-names> specify the subscript names. The <parent-name> is the data name with which the subscript is associated. The first three integers associated with the argument specify the lower bound, upper bound and increment, respectively, associated with the subscript name. The last integer specifies the dimension of the parent name with which the subscript name is associated.

The default values for the lower bound and upper bound are the corresponding bounds of the subscripted data name specified in the dd-statement. The default value for the increment is 1. The default value for the dimension (fourth integer) is also 1.

Example:

```
    I IS SUBSCRIPT(1,9,2)
```

This statement specifies a subscript name, I, the range of which is given by the set of indices (1,3,5,7,9). If an assertion uses I as a subscript, and no "for-clause" is specified for that assertion, then the values specified in the subscript statement is used as the range for the subscript in the assertion. If there is a for-clause in an assertion, then the range specified in the for-clause overrides the range given in the subscript statement. For example, the above subscript statement along with the following assertion:

```
    B(I) = B(I-1)
```

is equivalent to the following assertion:

```
    FOR I = 1 TO 9 BY 2 B(I) = B(I-1)
```

4.4.8 Summary of Parameters used in dd statements.

Table 4.1 gives a list of all parameters (except parent-name, repetition specification, and field specification) that can be used in different MODEL statements. Second column in the table specifies the statements in which the parameter can be specified. The third column specifies the abbreviations and alternatives for the parameter. Fourth column specifies the list of allowed values for the parameter (the abbreviations for these values have not been shown). Fifth column specifies the default values for the argument. And finally, the last column specifies the way to use the parameter, either as a positional parameter or as key-word parameter. If "EITHER" is specified in the last column, either form can be used. For example:

```
ORG=ISAM
```

can be alternately specified as:

```
ISAM
```

TABLE 4.1

PARAMETER	ALLOWED IN	ABBREVIATIONS/ ALTERNATIVES	ALLOWED VALUES	DEFAULT	KEYWORD/ POSITIONAL
BLOCKSIZE	MEDIA AND FILE STATEMENTS	BLKSIZE BS	<INTEGER>	RECORDSIZE * MAX(1,6400/ RECORDSIZE)	EITHER
CHAR_CODE	MEDIA AND FILE STATEMENTS	CC	EBCDIC BCD ASCII	EBCDIC	KEYWORD
DENSITY	MEDIA AND FILE STATEMENTS	DEN	200 556 800 1600	1600	KEYWORD
DISPOSITION	MEDIA AND FILE STATEMENTS	DISP	OLD SHR NEW MOD	OLD	EITHER
DSNAME	FILE STATEMENTS	DSN FILE_NAME	STANDARD DATA DATA SET NAME	-	KEYWORD
EXT_NAME	MEDIA AND FILE STATEMENTS	EL ELABEL VOLSER	<NAME>	XXXXXX	KEYWORD
INT_NAME	MEDIA AND FILE STATEMENTS	IL ILABEL	<NAME>	XXXXXX	KEYWORD
KEY	FILE, RECORD AND GROUP STATEMENTS	-	<NAME>	-	KEYWORD
LINESIZE	FILE AND MEDIA STATEMENTS	WIDTH LINE_WIDTH	<INTEGER>	120	KEYWORD

TABLE 4.1 (CONTD.)

PARAMETER	ALLOWED IN	ABBREVIATIONS/ ALTERNATIVES	ALLOWED VALUES	DEFAULT	KEYWORD/ POSITIONAL
NO_TRKS	FILE AND MEDIA STATEMENTS	TRKS TRACKS	7 9	9	KEYWORD
ORGANIZATION	FILE AND MEDIA, STATEMENTS	ORG	SAM ISAM DIRECT REG2 REG3	SAM	EITHER
PAGESIZE	FILE AND MEDIA STATEMENTS	LINES_PER_PAGE	<INTEGER>	60	KEYWORD
PARITY	FILE AND MEDIA STATEMENTS	PAR	ODD EVEN	ODD	KEYWORD
RECORD_FORMAT	FILE AND MEDIA STATEMENTS	RF RECFM	FIXED FB VARIABLE VB VS UNDEFINED	UNDEFINED	EITHER
RECORDSIZE	FILE AND MEDIA STATEMENTS	RL LRECL RECSIZE	<INTEGER>	LENGTH OF RECORD	KEYWORD
SPACE	FILE STATEMENTS	-	STANDARD SPACE SPECIFICATION FOR EXAMPLE: (TRK,100,5,2,RLSE)	(TRK,10,1,1)	KEYWORD
START_FILE	FILE STATEMENTS	FILE#	<INTEGER>	1	KEYWORD
TAB	FILE AND MEDIA STATEMENTS	-	A LIST OF <INTEGER>S	(10,20, ... 120)	KEYWORD

TABLE 4.1 (CONTD.)

PARAMETER	ALLOWED IN	ABBREVIATIONS/ ALTERNATIVES	ALLOWED VALUES	DEFAULT	KEYWORD/ POSITIONAL
TLABEL	FILE AND MEDIA STATEMENTS	TL	SL NL BLP	SL	KEYWORD
UNIT	FILE AND MEDIA STATEMENTS	DEVICE	PUNCH CARD TAPE DISK 2314 2311 2305 TERMINAL OLS	OLS	EITHER

4.5 Assertions Section.

This section contains the following types of statements:

(1). Assertion statements,
(2). Source Variable Statements,
(3). Target Variable Statements,
(4). Function Statements,
(5). Initial statements,
(6). Test statements, and
(7). Solution Method statements.

The statement types (2), (3), (5), (6) and (7) above are essentially associated with the simultaneous equations. Statement types (2) and (3) can be specified even for simple assertions; however, they are not necessary if the following rule is satisfied:

All the variables on the LHS of an assertion are the target variables; and all variables on the RHS of the assertion (and in the boolean clause) are the source variables.

The above type of statements are described in detail in the following.

4.5.1 Assertions.

Purpose: To specify relationships between data names used in the module specification.

Syntax: Except for the <if-or-for-clause>, described below, the syntax of an assertion statement is almost the same as the syntax of an assignment statement in PL/1. The formal syntax of assertions is given below:

```
<assertion-statement>::= [<qualified-name>:] <assertion>

<assertion>::= <if-or-for-clause> <assertion>
                  [ELSE <assertion> ]
         |  <variable> = <arith-expression>

<if-or-for-clause>::= <if-or-for> <for-expression>
                   [<then-or-let>]

<if-or-for>::= IF | FOR

<for-expression>::= <simple-for> [<and-or-or>
                   <for-expression>]

<simple-for>::= <arith-expression>
         [<from-or-eq> <arith-exoression>
         [<rest-of-bool>
         [ BY <arith-expression> ]]]

<from-or-eq>::= FROM | <rel-opr>

<rel-opr>::= < | <= | > | >= | = | =

<rest-of-bool>::= TO <arith-expression>
         | <rel-opr> <arith-expression>

<and-or-or>::= &  | |

<then-or-let>::= THEN | LET

<arith-expression>::= [<mon-opr>] <relational>

<mon-opr>::= + | - |

<relational>::= <primary> [<operators> <relational>]

<operators>::= <rel-opr> | <diadic-opr>  | <and-or-or>

<primary>::= ( <arith-expression>)
        | <integer>
        | <float-data>
```

```
            | <character-string>
            | <bit-string>
            | <variable>

<variable>::= <qualified-name> [<subscripts>]

<subscripts>::= (<arith-expression>
                [,<arith-expression>]*)

<diadic-opr>::= * | / | + | - | ** | ||

<character-string>::= ' <any-string-of-characters> '

<bit-string>::= ' <bits>[<bits>]* 'B

<bits>::= 0 | 1

<float-data>::= [<sign>] <digit-string> [E [<sign>]
                    <string-of-digits>]

<sign>::= + | -

<digit-string>::= <str-of-digits> [.[<str-of-digits>]]
                    | . <str-of-digits>

<str-of-digits>::= <digits> [<digits>]*
```

In the above BNF specification, the assertion is defined recursively. There is no restriction imposed on the depth of recursion.

Semantics:

The qualified name in <assertion-statement> specifies the name of the assertion. If this name is not specified, then the system automatically supplies a default assertion name.

In general, an assertion has one of the following forms:

```
(1).   <data-name> = <expression>

(2).   IF <boolean-expression> THEN
                 <data-name> = <expression>
          [ELSE <data-name> = <expression>]

(3).   FOR <for-clause> <data-name> = <expression>
```

Essentially, an assertion statement can be divided into two parts: (1) <if-or-for-clause> and (2) <simple-assertion>.

<u>if-or-for-clause</u> specifies either a boolean expression, just as in the case of PL/1, or a set of subscript ranges for the subscripts used in the assertion. The operators '|' and '&' can be used to compose compound boolean expressions or compound subscript ranges. if-or-for-clause can be specified in one of the following ways:

(1). A simple variable name. In this case, the variable name must be defined as a bit string data.

(2). A boolean expression (or a logical expression) using any data name in the module specification.

(3). Any combination of (1) and (2) with the use of operators '|', '&' and ''.

(4). A simple subscript specification of the form:

```
FOR I = 10
```

(5). A subscript range specification of the form:

FOR I = $\underline{l}$ to $\underline{u}$ [By $\underline{c}$]

(6). A subscript range specification of the form:

FOR $\underline{l}$ <= I <= $\underline{u}$ [BY $\underline{c}$]

where in (4), (5) and (6), I is a subscript name, and $\underline{l}$, $\underline{u}$ and $\underline{c}$ are the lower bound, upper bound and increment, respectively, associated with the subscript.

(7). Any combination of (4), (5) and (6) with the use of operators '|' and '&'.

(8). Any combination of (1) thru (7) with the use of the operators '|' and '&'.

The first three types in the above list specify boolean expressions associated with the assertion and the types (4), (5), (6) and (7) specify the subscript ranges associated with the subscripts used in the assertion.

Some examples of boolean expressions are:

```
A = 'NULL'
A > 1000 | B < 20
(A = B| C = D) & (E = 1)
```

Some examples of for-clauses are:

```
I = 1 TO 9 BY 2
I = 1 TO 10 | I = 20 TO 30
I = 1 TO 10 & J = 1 TO 5
```

Simple Assertions specify the relationships between the source and target variables used in the assertion. It is in the form of an equation; the LHS of the equation specifies the target variable in the assertion and the RHS specifies the source variables used in the assertion. LHS of the equation must be a single data name, which may be subscripted. RHS of the equation is an expression composed of variable names (which may be subscripted), constants and functions. Some of the builtin functions like, SUBSTR, can also appear on the LHS of the equation. This situation is explained in later sections.

Some examples of simple assertions are:

```
        A = B + C
        A = 'THAT IS '|| 'RIGHT'
        ABC(10) = 100
        SUBSTR(X,Y,4) = 'JUNK'
```

In the last example, the data name X is assumed to be the target variable and Y as the source variable.

<character-string> (in the production <primary>) consists of characters enclosed in quotation marks. If a single quotation mark itself is to be enclosed in the character string, then the quotation mark must be written as two single quotation marks with no intervening blanks. For example, the character string constant:

```
'SHAKESPEARE''S ''''HAMLET'''''
```

indicates the character string: SHAKESPEARE'S ''HAMLET''.

<u><bit-string></u> consists of strings of digits 0 or 1 enclosed in single quotation marks; the letter B must follow the string immediately.

<u><float-data></u> recognizes decimal floating point data which consists of a string of digits (with an optional decimal point) and an optional exponent. Some valid floating point data constants are:

```
12.34E-5
.12E-10
12.000
12.34
```

<u>Example1</u>:

```
ASS1: IF A = B THEN C = D ELSE C = E(X)
```

This assertion contains a boolean expression "A=B" and two simple assertions "C=D" and "C=E(X)". The target variable associated with this assertion is C and the source variables are A, B, D, E and X. The name ASS1 is the assertion name.

<u>Example2</u>:

```
FOR I = 1 TO 10 A(I) = A(I-1)
```

This assertion contains a for-clause specifying the range of subscript I used in the simple assertion "A(I)=A(I-1)". Source variable of this assertion is A(I) for I = 0 to 9, and the target variable of the assertion is A(I) for I = 1 to 10. Assertions with subscripted variables are described in the later sections.

Example3:

```
    IF C = B & I = 1 TO 10 THEN A(I) = A(I-1)
```

This assertion contains a boolean expression "C=B" and a for-clause "I= 1 to 10". The target variable of this assertion is A(I) for I = 1 to 10; and the source variables are C, B and A(I) for I = 0 to 9.

4.5.2 Source Variable Statements.

Purpose: To specify the source variable names used in an assertion. This statement is needed only in defining a set of simultaneous statements, or when the rule given in section 4.5 is not applicable.

Syntax:

```
<source-var-statement>::= <assertion-name>:
      <keyword1> : <list-of-variables>
```

where:

```
<keyword1>::= SOURCE | SOU

<list-of-variables>::= <variable> [,<variable>]*
```

and the item <variable> has the same syntax as the corresponding item in assertion statement (see sub-section 4.5.1).

Semantics: The variables given by <list-of-variables> are the source variables for the assertion given by <assertion-name>. As in assertion statement, an assertion name can be a qualified name. If no source variable statement is specified for an assertion, then all the variables in the boolean expression and RHS of the assertion are assumed to be the source variables.

Example:

```
ASS1: SOURCE: A,B,C
X = A + B + C(1234)
```

The first statement in the above example specifies that the variables A, B and C used in the assertion ASS1 are the source variables. The second statement specifies the assertion itself.

4.5.3 Target Variable Statements.

Purpose: To specify the target variables used in an assertion. This statement is needed only in defining a set of simultaneous statements, or when the rule given in section 4.5 is not applicable.

Syntax:

```
<target-var-statement>::= <assertion-name>:
      <keyword1> : <list-of-variables>
```

where:

```
<keyword1>::= TAR | TARGET
```

and the item <list-of-variables> has the same syntax as the corresponding item in source variable statement (see section 4.5.3).

Semantics: The variables given by the <list-of-variables> are the target variables of the assertion, the name of which is given by <assertion-name>.

If no target variable statement is specified for an assertion, then all the variables on the LHS of the assertion are assumed to be the target variables.

Example:

```
     ASS1: TARGET: X
```

This statement specifies that the variable X is the target of the assertion ASS1.

4.5.4 Function Statements.

Purpose: To specify the function names used in an assertion, or in the model specification. This statement is used only in the definition of a set of simultaneous equations.

Syntax:

```
<function-statement>::= [<assertion-name> : ]
        <keyword1> : <list-of-functions>
```

where:

```
<keyword1>::= FUNCTION | FUN

<list-of-functions>::= <name> [(<arguments>)]

<arguments>::= <field-desc> [,<field-desc>]
```

The item <field-desc> has the same syntax as the corresponding item in field statement described in section 4.4.5.

Semantics: The names in the <list-of-functions> specify the function names used in the module specification (if no assertion name was specified), or in the particular assertion name of which is given by <assertion-name>.

The <arguments> associated with each function name specify the attributes of the arguments of the function. As in the case of PL/1, these arguments are required if the function is in a library (that is, not defined by the user in the module specification) and if the attributes of the arguments of the function differs from the attributes specified in the usage of that function. The attributes of an argument is same as the field type of a data name which is described by the item <field-desc> in section 4.4.5.

Example:

```
      FUNCTION: COMPUTE(CHAR(4))
```

This statement specifies the attributes of the one and only one argument of the function COMPUTE which is a character string of length 4.

4.5.5 Initial Statements.

Purpose: This statement is used to specify the initial values of the source variables used in a set of simultaneous equations.

Syntax:

```
<initial-statement>::= <assertion-name> :
              <keyword1> : <simple-assertions>
where:
<keyword1>::= INIT | INITIAL
<simple-assertions>::= <variable> = <arith-expression>
                        [,<variable> = <arith-expression>]*
```

The items <variable> and <arith-expression> has the same syntax as the corresponding statements in assertion statements described in sub-ection 4.5.1.

Semantics: The set of values specified by each

<arith-expression> in the <simple-assertions> specifies the set of initial values for the variables specified on the LHS of the corresponding simple assertion. The variables used in the LHS of each simple assertion must be used in the assertion, the name of which is given by <assertion-name>.

Example:

```
    SIM: INIT: X0=1, Y0=0
```

The above statement specifies the initial values of 1 and 0 for the variables x0 and y0 respectively, which are used in the assertion SIM.

4.5.6 Test Statements.

Purpose: These statements are used to specify the test values associated with each variable in a set of simultaneous equations. These test values are used to terminate the iteration in the solution method adopted for solving the set of simultaneous equations.

Syntax:

```
<test-statement>::= <assertion-name> :
        TEST : <simple-assertions>
```

where <simple-assertion> has the same syntax as the corresponding term in initial statement described in section 4.5.5.

Semantics: The <arith-expression> in each simple assertion specify the test values for the corresponding variable on the LHS of the simple assertion.

Example: SIM: TEST : X0= .0002

This example specifies the test value of .0002 for the variable X0 for the termination of iteration in solving the assertion SIM.

4.5.7 Solution Method Statements.

Purpose: These statements describe the solution method to be used to solve the set of simultaneous equations.

Syntax:

```
<solution-stmt>::= <assertion-name> :
      <keyword1> : <parameters>
```

where:

```
<keyword1>::= SOL | SOLUTION
<parameters>::= <name> [,<arith-expression>]*
```

and the term <arith-expression> has the same syntax as the corresponding item in assertion statement described in section 4.5.1.

Semantics: The name in <parameters> is the solution method to be used for the assertion, the name of which is given by <assertion-name>. If second parameter is specified, then it specifies the maximum number of iterations to be performed in the solution method. Only two parameters is currently being used.

<u>Example</u>:

```
      ASS: G_S, 10
```

The above statement specifies that Gauss-Seidel solution method be used and the maximum number of iterations to be performed is 10.

4.6 <u>Help Statement</u>.

In addition to the statements described in the earlier sections, one additional statement is allowed in MODEL. This statement, called HELP statement, is used in aiding the user by displaying the syntax any statement in MODEL, or by displaying the description of an error code reported by the processor. The semantics of HELP statement is described in detail in the next chapter. In the following, the purpose and syntax of the statement is given.

<u>Purpose</u>: To aid the user with the description of an error code or the description and syntax of a particular MODEL

statement.

Syntax:

```
 <help-statement>::= HELP [<argument>]
 <argument>::= MODEL |
               SYNTAX |
               <statement-type> |
               <error-code>
```

4.7 Keywords in MODEL.

Currently, only the following keywords are used in MODEL: ABSENT, ENDGRP, ENDFILE, EXIST, LENGTH, POINTER, and SUBSET.

All the above keywords are used as the highest qualifier for a data name, like in, POINTER.MAST, SUBSET.REPORT, and so on. The significance of each of the above keywords is briefly described in the following.

4.7.1 The Keyword ABSENT.

The keyword absent is associated with a file name, group name or a record name. It is used to detect the "absent" condition of the data name that follows it. For example, if MAST is a file name, then ABSENT.MAST is a bit variable that is equivalent to the following on-unit of PL/1:

```
     UNDEFINEDFILE(MAST)
```

Similarly, if REC is a record in an ISAM file, MAST, then

ABSENT.REC is also a bit variable which is equivalent to the following on-unit of PL/1:

KEY(MAST)

4.7.2 <u>The Keyword ENDGRP</u>.

The keyword ENDGRP is used to terminate an iteration corresponding to a repeating data name. For example, consider the set of MODEL statements shown in Figure 4.8.

```
G1 IS GROUP(*);
   G2 IS GROUP(G1,(*));
      F IS FIELD(G2);

T IS INTERIM(*);
IF F(I) = F(I+1) THEN ENDGRP.G2 = '1'B;
     ELSE ENDGRP.G2 = '0'B;
T = ANY(F);
```

Figure 4.8. An example using the keyword ENDGRP.

If the sequence of elements in G1 is given by:

10 12 12 12 13 13 16 20 20

then, the group G1 repeats 5 times and the five groups in G1 are given by the following sets of elements: (10), (12,12,12), (13,13), (16) and (20,20). T will therefore, contain the set of elements: (10,12,13,16,20).

4.7.3 <u>The Keyword ENDFILE</u>.

The keyword ENDFILE is associated with a file name to detect end of that file. For example, if MAST is a file, then ENDFILE.MAST is a bit variable which is equivalent to the following on-unit:

```
        ENDFILE(MAST)
```

4.7.4 The Keyword EXIST.

The keyword EXIST is used to indicate the size of a repeating data name. For example, for the set of statements in Figure 4.8, the user could explicitly give an expression for EXIST.G1 or EXIST.G2, like:

```
         EXIST.G2 = 10;
```

In this case the Group is assumed to be repeating 10 times.

4.7.5 The Keyword LENGTH.

The keyword LENGTH is used to indicate the length of a string. For example, if the field X is defined by the following statement:

```
        X IS FIELD (CHAR(*));
```

Then, LENGTH.X can be used to indicate the actual length of X.

4.7.6 The Keyword POINTER.

The keyword POINTER is used to select a particular member of a repeating group or record. Most frequent use of this keyword is to select a record of a keyed file. For example, if MAST is a keyed file, with the key name PRODUCT_NUMBER, then the following assertion:

```
    POINTER.MAST = TRANSACTION.PRODUCT_NUMBER;
```

will select that record in file MAST, which corresponds to the PRODUCT_NUMBER in the file TRANSACTION.

4.7.7 The Keyword SUBSET.

The keyword SUBSET is used with a file name, to select a subset of records in that file. For example, if the following assertion was specified:

```
     IF <logical-expression> THEN SUBSET.MAST = SELECTED;
                         ELSE SUBSET.MAST = SELECTED;
```

And, if MAST is a source file, then a record is read from MAST only if SUBSET.MAST is set in the above assertion. However, if MAST is a target file, then a record is written to that file only if SUBSET.MAST is set in the above assertion.

4.8 An Illustrative Example.

We will resort to an example to illustrate the use of MODEL. This example is taken from the book by McCracken and Garbassi (McC70). It envisages an environment of a sales organization with a number of salesmen and extensive inventory. The user of MODEL is envisaged to be a sales analyst wishing to specify the requirement for sales reporting. Figure 4.9 illustrates in block diagram form, the requirement which is a subset of the case study in the reference (McC70). The program module to be produced by the MODEL Processor, as shown at the center of Figure 4.9, is named RUN_1. It will maintain year-to-date sales in a file named MASTER. Sales information, consisting of PRODUCT_NUMBER, QUANTITY, SALESMEN and DISTRICT is obtained from the file nameed TRANSACTION. The REPORT_1 file consists of monthly sales of each product. The case study in the reference includes an additional report which is omitted here for sake of simplicity.

The MODEL specification for the program module RUN_1 is shown in Figure 4.10. The first three statements in this figure represent the header section. Statements 4 thru 10 define the file MASTER, and its components. Statements 11 thru 18 define the file TRANSACTION and its components. Statements 19 thru 31 define the file REPORT_1 and its components. The subscript name J is associated with the repeating data name TRANS on statement 32. And finally, the assertions in statements 33 thru 39 define the relations between the data items.

The TRANSACTION file is regarded as consisting of groups of records. Each group corresponds to a particular PRODUCT_NUMBER. The assertion on statement 39 indicates the upper bound of repetition for the group TRANS. In this case, the upper bound of the subscript J is equal to the number of transaction records corresponding to the same PRODUCT_NUMBER. This type of relation is defined by the variable name that begin with the keyword ENDGRP (see also section 4.6).

The function SUM used in statement 33 is used to obtain the sum of product totals. Note that SUM is a reduction function.

The function ANY used in statement 35 is also a reduction function. This function is required, because, we need to read only one Master record for every group of transactions (that have the same PRODUCT_NUMBER).

The above example will be referred in the discussion of the MODEL Processor in the chapters that follow. In particular, the generated reports from the specification of RUN_1 in Figure 4.10 will be described in Chapter 10.

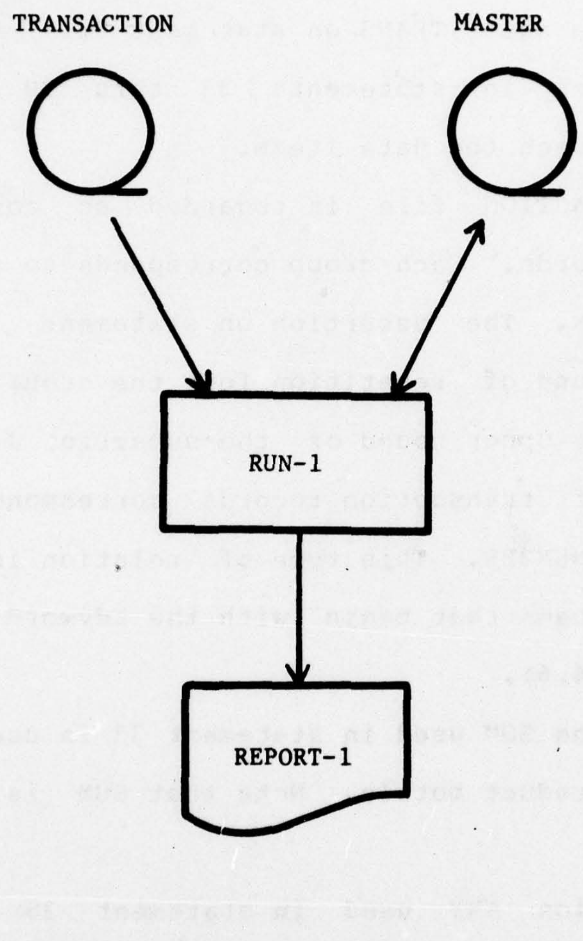


Figure 4. 9. An Overview of a simple Data Processing Problem.

```
STATEMENT
NUMBER
    1      MODULE: RUN_1;
    2        SOURCE: TRANSACTION,MASTER;
    3        TARGET: REPORT_1,MASTER;
    4        TAPE IS MEDIA(TAPE);
    5        MASTER IS FILE(TAPE,SAM,KEY=PRODUCT_NUMBER);
    6        MAST IS RECORD(MASTER);
    7        PRODUCT_NUMBER IS FIELD(MAST,PIC'X(5)');
    8        UNIT_PRICE IS FIELD(MAST,PIC'999V99');
    9        YTD_SALES IS FIELD(MAST,PIC'9(5)V99');
   10        FILLER IS FIELD(MAST,CHAR(63));
   11
   11      TRANSACTION IS FILE(TAPE,SAM,KEY=PRODUCT_NUMBER);
   12        PRODUCT IS GROUP(TRANSACTION,(*));
   13        TRANS IS RECORD(PRODUCT,(*));
   14        PRODUCT_NUMBER IS FIELD(TRANS,PIC'X(5)');
   15        QUANTITY IS FIELD(TRANS,PIC'999');
   16        SALESMAN IS FIELD(TRANS,PIC'XX');
   17        DISTRICT IS FIELD(TRANS,PIC'X');
   18        FILLER IS FIELD(TRANS,CHAR(69));
   19
   19      PRINTER IS MEDIA(PRINT);
   20        REPORT_1 IS FILE(PRINTER,SAM,KEY=PRODUCT_NUMBER);
   21        PAGE IS GROUP(REPORT_1);
   22        HEADING IS RECORD(PAGE);
   23        PRODUCT_ID IS FIELD(HEADING,PIC'X(10)');
   24        PRODUCT_TOTAL IS FIELD(HEADING,PIC'X(14)');
   25        PRODUCT_YTD IS FIELD(HEADING,PIC'X(14)');
   26        REP_1 IS RECORD(PAGE,(*));
   27        FILLER1 IS FIELD(REP_1,PIC'X(5)');
   28        PRODUCT_NUMBER IS FIELD(REP_1,PIC'X(5)');
   29        FILLER IS FIELD(REP_1,PIC'X(5)');
   30        PRODUCT_TOTAL_FOR_MONTH IS FIELD(REP_1,PIC'ZZZZZ.99');
   31        YTD_SALES IS FIELD(REP_1,PIC'ZZZZZ.99');
   32        J IS SUBSCRIPT (TRANS);
   33
   33        PRODUCT_TOTAL_FOR_MONTH = SUM(QUANTITY(J)*UNIT_PRICE);
   34         TARGET.YTD_SALES = SOURCE.YTD_SALES+PRODUCT_TOTAL_FOR_MONTH;
   35        POINTER.MAST = ANY( PRODUCT.PRODUCT_NUMBER);
   36        PRODUCT_ID = 'PRODUCT';
   37        PRODUCT_TOTAL = 'MONTH SALES';
   38        PRODUCT_YTD = 'YTD SALES';
   39        IF TRANS.PRODUCT_NUMBER(J) ¬= TRANS.PRODUCT_NUMBER(J+1) THEN
   39          ENDGRP.TRANS = TRUE
   39          ELSE ENDGRP.TRANS = FALSE
```

Figure 4.10. **MODEL Specification for the data Processing problem shown in Figure 4.9.**

CHAPTER 5

SYNTAX ANALYSIS

This chapter briefly describes the functions of the syntax analyzer in the MODEL Processor. The Syntax Analysis Program (SAP) for MODEL is generated by the meta processor called Syntax Analysis Program Generator (SAPG). The SAPG used here was developed by the DDL Project at the Moore School, and was found invaluable in the development of the MODEL Processor. In the following, SAPG, SAP and the supporting routines used by SAP are briefly described.

5.1 Generation of SAP using SAPG and EBNF.

The overview of SAPG is shown in Figure 5.1. As shown, SAPG produces SAP from the specification of the MODEL language in the meta language - "Extended Backus Normal Form" (EBNF). The EBNF includes the traditional concepts of BNF and the following extensions:

(1). The meta symbols [and] are used to represent optionality, and

(2). An asterisk immediately following [] represents repetition of the item inside the brackets zero or more times.

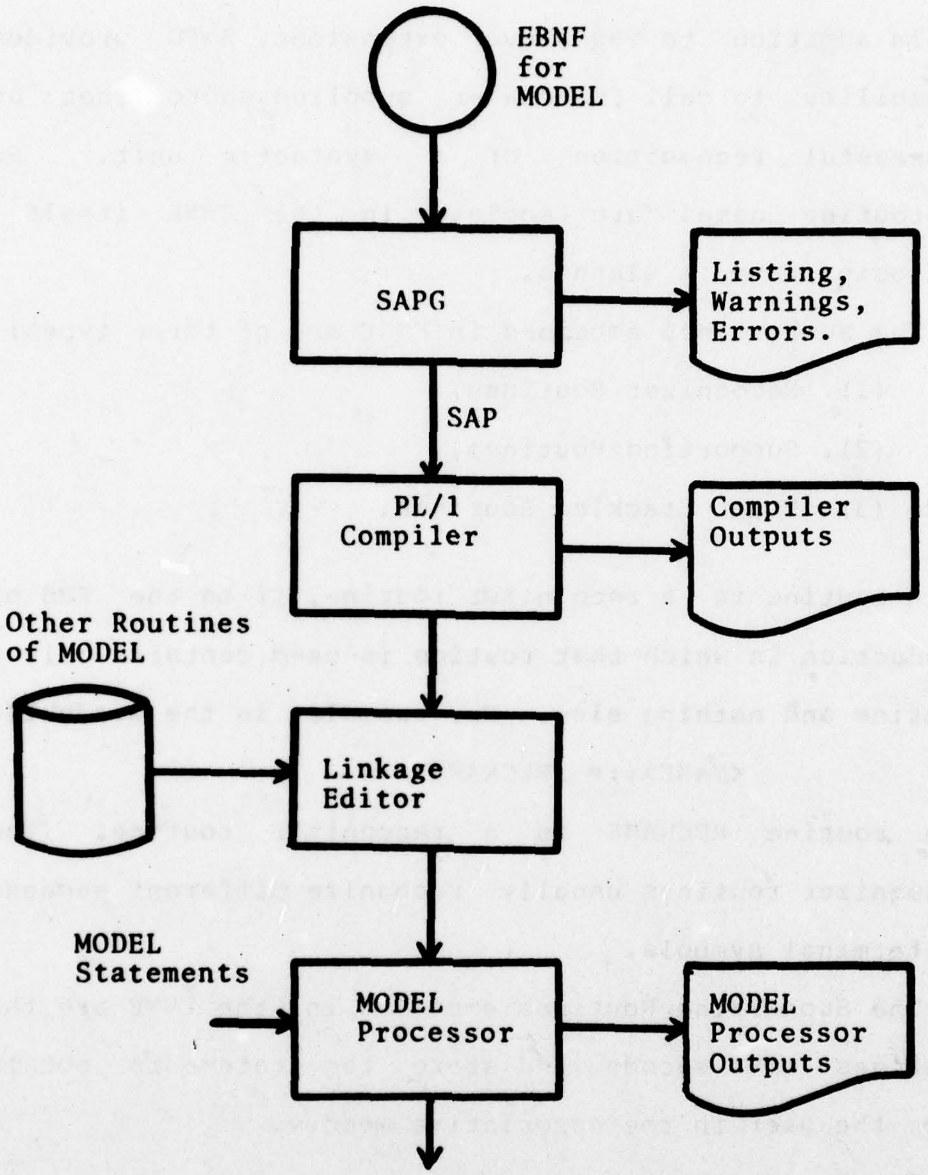


Figure 5.1. Overview of SAPG, SAP and MODEL Processor.

In addition to the above extensions, SAPG provides a capability to call some user supplied subroutines upon successful recognition of a syntactic unit. Such subroutine names are enclosed in the EBNF itself by enclosing them in slashes.

The subroutines embedded in EBNF are of three types:

(1). Recognizer Routines,

(2). Supporting Routines,

and (3). Error Stacking Routines.

A routine is a recognizer routine, if on the RHS of a production in which that routine is used contains only the routine and nothing else. For example, in the production:

<NAME>::= /RECNAME/

the routine RECNAME is a recognizer routine. These recognizer routines usually recognize different sequences of terminal symbols.

The Supporting Routines embedded in the EBNF are those routines that encode and store the statements obtained from the user in the associative memory.

The Error Stacking routines are used to stack error codes. One error code is stacked for each expected token in the input. If an expected token was found, the SAP automatically pops the error code in the top of the error stack. Otherwise, the error code in the top of the stack is used to inform the user that a corresponding token was missing in the input.

It is difficult to distinguish between a supporting routine and an error stacking routine in an EBNF. However, the difficulty has been eliminated in the EBNF for MODEL (see section 5.8), by using the following convention: the name of every error stacking routine starts with the letter 'W'.

Because SAPG has been described earlier in many reports, the generation of SAP using SAPG will not be described further. The reader intending to use SAPG is advised to refer the original reports (RAM73, FRE72, RIN76).

A detailed flowchart of the syntax analysis phase is shown in Figure 5.2. Some of the sections in this flowchart are described in the following.

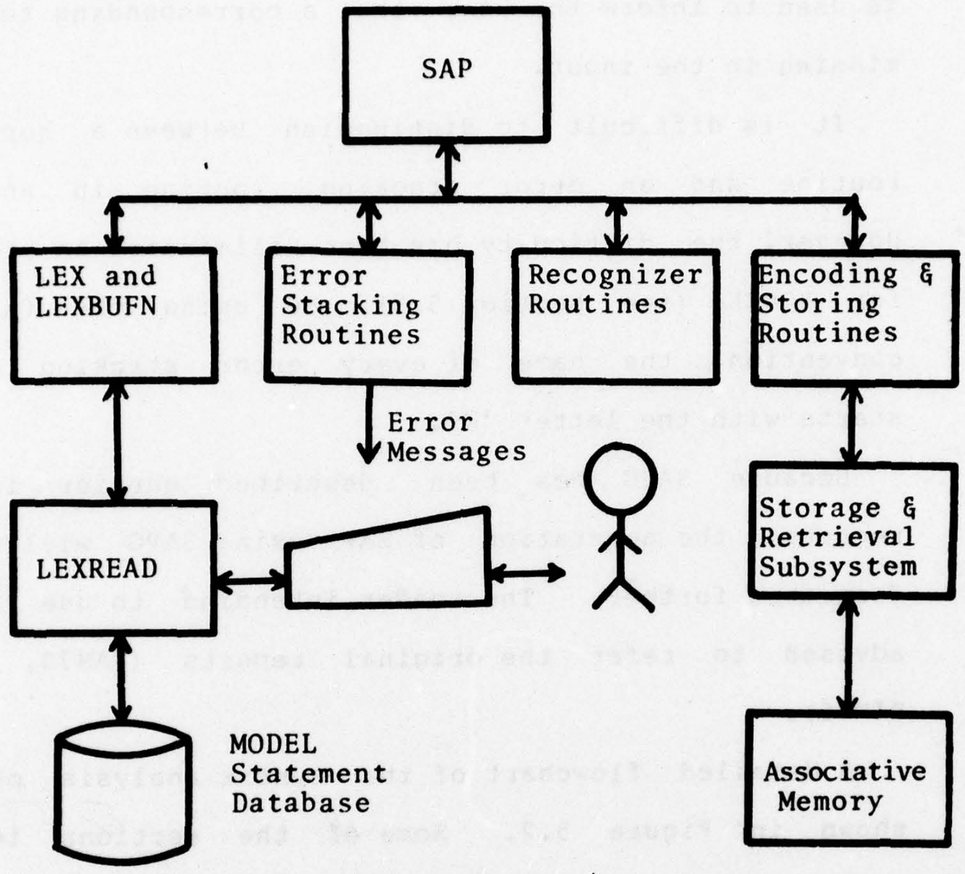


Figure 5.2. Syntax Analysis in MODEL Processor.

5.2 Lexical Analyzer.

The purpose of the Lexical Analyzer is to scan for syntactic units or "tokens", using such delimitors as blanks and certain punctuation marks, and return them to the calling program for syntactic checking. The input statements are obtained from the terminal or from the MODEL statement database (if REFER statements were used). The lexical analyzer is also described in the earlier reports (RAM73, FRE72). However, one additional feature has been added to the current system. This feature is the lookahead feature, which allows a supporting routine to look ahead (to a certain extent) of the current token. This look-ahead facility provides back-tracking in SAPG, which is otherwise not possible. To illustrate this point, consider the following three production rules:

```
<X>::= <Y> | <Z>
<Y>::= A B ...
<Z>::= A C ...
```

Because there is no back-tracking facility in SAPG, the strings in <Z> cannot be recognized using the above productions (see also the references RAM73, FRE72 and RIN76). However, using the look-ahead facility, it is possible to check the next token (while recognizing production <Y>), and if it not B, then the production <Y> is disabled. Therefore, the strings in <Z> can be recognized. The function LEXBUFN(N) is used to look-ahead to the N th token. To illustrate the use of the routine LEXBUFN, let us rewrite the above three productions as follows:

```
<X>::= <Y> | <Z>

<Y>::= <CHEKY> A B ...

<Z>::= A C ...

<CHEKY>::= /CHEKY/
```

The recognizer routine CHEKY used in the last production rule can be written as follows:

```
CHEKY: PROC RETURNS(BIT(1));
DCL LEX ENTRY;
DCL LEXBUFN ENTRY(BIN FIXED) RETURNS(CHAR(31)VAR);
DCL LEXBUFF CHAR(31) VAR EXT;
   CALL LEX;
  IF LEXBUFF = 'A' THEN
      IF LEXBUFN(1) = 'B' THEN RETURN('1'B);
         ELSE RETURN('0'B);
      ELSE RETURN('0'B);
  END CHEKY;
```

4.1 Error Stacking Routines.

One disadvantage of SAPG is its inability to produce good error diagnostics automatically when an expected token is missing in the input. User has to provide the capability bv embedding the error stacking routines in the EBNF. In general, an error routine must be specified before every syntactic unit. The function of the error stacking routine is to stack an error code on the top of an error stack, and if the expected token (syntactic unit) was recognized successfully, then the SAP simply pops the error code on the top of the error stack and scaninning is continued. If the expected token was not found in the input, an error message associated with the error code is reported to the user; and the statement till the error is discarded.

5.4 Recognizer Routines.

These routines are supplementary to SAP and performs syntactic recognition of some special tokens like:

(1). data names (routine RECNAME),

(2). Integer and floating point data constants (routines RECINT and RECFLOT),

(3). Charater and Bit strings (routine LEXCSTR),

and (4). special operators (routines RECOR, RECEXP, etc.).

These recognizer routines check for a generic class of syntactic units, and in general, it is very difficult to represent the function of a recognizer routine as a production in EBNF. This is because, the tokens returned by the lexical analyzer are not single characters, but a string of characters separated by some delimiter. In some cases, it is impossible to represent a syntactic unit by a production rule. For example, if we want to recognize the operator '||', and if we write the following production rule:

    <RECEXP>::= ||

then, SAPG will be totally confused because, it treats '|' as a meta-symbol used to separate alternative syntactic units. Therefore, such a syntactic unit must be recognized using a recognizer routine. For the above example, the production rule will be:

    <RECEXP>::= /RECEXP/

and the routine RECEXP will look as follows:

```
RECEXP: PROC RETURNS(BIT(1));
DCL (LEX,LEXENAB) ENTRY;
DCL LEXBUFF CHAR(31) VAR EXT;
  CALL LEX;
  IF LEXBUFF = '||' THEN DO;
      CALL LEXENAB;
      RETURN('1'B);
  END;
      ELSE RETURN('0'B);
END RECEXP;
```

5.5 Encoding and Storing Routines.

These routines essentially perform the following functions:

(1). Check local semantics of different attributes associated with a particular statement.

(2). Encode the user statements.

and (3). Store the encoded statements in the associative memory.

Local semantic check is performed on some tokens, like checking the range of an integer constant, validating the dimension specified in a field, group or interim statement, and so on.

Encoding routines encode some of the attributes of a MODEL statement into an internal representation. The encoded data is stored in the data area of the storage entry for the statement. The different encodings used for each statement is described in the next chapter.

The storing routines are used to store the data names used in a MODEL statement (and the encoded data associated with it) in the associative memory. All these routines use the STORE routine which is a part of the storage and retrieval sub-system described in the next chapter.

5.6 HELP - A Teaching Aid During Syntax Analysis.

In an effort to make the MODEL language usable by a novice user, a teaching aid, HELP, is provided during syntax analysis. This aid can be used to obtain a brief description of the syntax of any statement in MODEL language, or to know the significance of an error message reported to the user. This teaching aid can be utilized by including a HELP statement in the input to SAP. As any other statement in MODEL, HELP statement also has a proper syntax. However, HELP statement is never stored in the associative memory, and any information requested by HELP statement is immediately displayed to the user. The formal syntax of HELP statement can be described simply by the following production rules:

```
   <help-statement>::= HELP [<argument>]
   <argument>::= MODEL
               | SYNTAX
               | <statement-type>
               | <error-code>
```

If the argument is MODEL, then a short list of all allowed statements in MODEL is displayed.

If the argument is SYNTAX, then syntactical conventions used in the description of syntax of MODEL statements are displayed.

The syntactic unit <statement-type> in the above production specifies type of statement, the syntactical description of which is desired. Currently, <statement-type> can be any one of the keywords described in the following table:

<statement-type>	describes the syntax of:
MODULE	Module statement
SOURCEF	Source File statement
TARGETF	Target File statement
REFER	Refer statement
MEDIA	Media statement
FILE	File statement
RECORD	Record statement
GROUP	Groutp statement
FIELD	Field statement
INTERIM	Interim statement
SUBSCRIPT	Subscript statement
ASSERTIONS	Assertions.

The syntactic unit <error-code> in the production rule of <argument> specifies the error code, the description of which is to be displayed. In this case, a short description of the error code is displayed on the terminal.

If no argument is specified in a HELP statement, then a short description of the error encountered most recently is displayed on the terminal. If no error was encountered, then the argument is assumed to be MODEL.

The use of HELP is demonstrated in the example shown in Figure 5.3. The listing for every use of HELP command is given in Appendix B.

```
help model

FOLLOWING TYPES OF STATEMENTS ARE ACCEPTED BY MODEL:
   (1). MODULE STATEMENT (MODULE).
   (2). SOURCE FILE STATEMENT (SOURCEF).
   (3). TARGET FILE STATEMENT (TARGETF).
   (4). REFER STATEMENT (REFER).
   (5). MEDIA DESCRIPTION STATEMENT (MEDIA).
   (6). FILE DESCRIPTION STATEMENT (FILE).
   (7). RECORD DESCRIPTION STATEMENT (RECORD).
   (8). GROUP DESCRIPTION STATEMENT (GROUP).
   (9). FIELD DESCRIPTION STATEMENT (FIELD).
   (10). INTERIM DESCRIPTION STATEMENT (INTERIM).
   (11). SUBSCRIPT DESCRIPTION STATEMENT (SUBSCRIPT).
   (12). ASSERTIONS (ASSERTION).

    TO DISPLAY THE SYNTAX AND DESCRIPTION OF ANY STATEMENT ENTER:
      HELP  <STMT-TYPE>
WHERE <STMT-TYPE> IS THE STATEMENT TYPE, WHICH IS ENCLOSED IN
PARENTHESES IN THE ABOVE LIST.
    FOR EXAMPLE, THE FOLLOWING STATEMENT:
      HELP FIELD
WILL DESCRIBE THE SYNTAX AND THE DESCRIPTION OF FIELD STATEMENT.

a is b + c;

A IS B + C;
  I
WEQUAL ERROR.  INVALID TEXT BEGINNING 'IS' IN LINE NUMBER 1.
               REST OF THE LINE TILL THE SEMICOLON IS DISCARDED.

help wequal

WEQUAL: EXPECTED TOKEN "=" IS MISSING.

a = b + c;
 ...
```

Figure 5.3. Illustrating the use of HELP command in MODEL. Entries in lower case are entered by the user. Rest are the responses by the system.

5.7 Referring MODEL Statements in the Library.

Once a module is stored in the library (MODEL statement database), any section of that module can be referenced in subsequent MODEL specifications. Such a reference is done by using the REFER statement, the syntax of which is given by:

```
<refer-statement>::= REFER <module-name> [.<section-name>]
```

where <module-name> is the name of the module and <section-name> is the name of the section (which may be qualified) in the module which is to be referenced. The organization of a module into sections is described in chapter 7. If the section name is not specified, then the whole module (or member in the library) is included in the current specification. Just as in the case of HELP statement, REFER statements are never stored in the associative memory, but as soon as a refer statement is successfully scanned, a switch is set in the lexical analyzer to read the input from the library, rather than from the terminal. When the reading from the library is complete, the switch is reset so as to obtain input from the terminal.

5.8 Complete EBNF (with SRC) for MODEL.

In the following, the complete EBNF for the MODEL system is given. The routines enclosed in slashes (/) are the subroutines that are called by SAP while recognizing the input statement corresponding to the production. A short description of each such routine is given in Appendix A.

EBNF FOR MODEL

```
<MODEL_SPECIFICATION>::= [<MODEL_BODY_STMT>/$CLRERR/]*
 /LEXFAIL/ <MODEL_SPECIFICATION>
<MODEL_BODY_STMT>::= /WUREC/ /SINITNM/
   <END_STMT>  [;]
 | <HELP> /SVCHELP/ [<NAME>/SVSHELP/]/SVXHELP/[;]
 | <REFER_STMT> [;] /LIBREAD/
 | <RSEC> /SVSNM1/ [<NAME>/SVSNM2/]/SVSNM3/
 | <MOD_REC> /WSCOL1/: [<VARIABLE>/SINC#ON/]/SSTMOD/[;]
 | <S_NAME> [<F_NAME>] /WSCOL1/ : <V_LIST> /SSTSRC/ [;]
 | <DDL_OR_ASSER_STMT> [;]
<END_STMT>::= END/ENDINPT/
<HELP>::= HELP | ?
<REFER_STMT>::= <RREFER>  [<RSEC>]
    [:] <VARIABLE> /SVREF/
    [,<VARIABLE> /SVREF/]*
<RREFER>::= REF | REFER | REFERENCE
<RSEC>::= SEC | SECTION | SECTIONS
<VARIABLE>::= /WVARN1/ <NAME> /SVAV1/ [. /WVARN2/ <NAME>
          /SVAV2/ ]*
<MOD_REC>::= MODULE | MOD
<S_NAME>::= SOURCE | TARGET | SOU | TAR
<F_NAME>::= F | FIL | FILE | FILES
<V_LIST>::= <VARIABLE> /SINC#ON/ [,<VARIABLE> /SINC#ON/]*
```

```
<DDL_OR_ASSER_STMT>::=  <VARIABLE> /SVANAM/
            <DDL_OR_ASSER_BODY>
<DDL_OR_ASSER_BODY>::= /WDDASS/ : <ASSERTION_BEG> /SVASST/
    |  [,<VARIABLE> /SINC#QN/]* [<IS>] <DATA_OR_ASSER>
<ASSERTION_BEG>::= <RECSTV> [ <RECVAR> ] /WSCOL1/:
              <AE_LIST>
  | <RECIT> /WSCOL2/ : <SAS_LIST>
  | <ASSERTION>
<RECSTV>::= INITIAL | INIT | INI | TEST | TES |
           SOURCE | SOU | TARGET | TAR |
           FUNCTION | FUN | SOL | SOLUTION
<RECVAR>::= V | VAR | VARS | VARIABLE | VARAIBALES
<RECIT>::= INITIAL | INIT | INI | TEST | TES
<AE_LIST>::=  <ARITH_EXPRESSION> /SVAAE5/ [,
       <ARITH_EXPRESSION>/SVAAE6/]* /SVAAE7/
<SAS_LIST>::=  <SIMPLE_ASSERTION> /SVAAE5/ [,
       <SIMPLE_ASSERTION>/SVAAE6/]* /SVAAE7/
<IS>::= IS | ARE | =
<ASSERTION>::=/WASS1/ /SVAAS1/ <IF_OR_FOR_CLAUSE>
          /SVAAS2/ <ASSERTION> /SVAAS3/
          [ELSE <ASSERTION> /SVAAS4/]
  |       <SIMPLE_ASSERTION> /SVAAS6/
<IF_OR_FOR_CLAUSE>::=  <IF_OR_FOR> <FOR_BOOL>
          [<THEN_OR_LET>]
```

```
<IF_OR_FOR>::= IF | FOR

<FOR_BOOL>::= /SVAFOR1/ <ARITH_EXPRESSION> /SVAFOR2/

<THEN_OR_LET>::= THEN | LET

<RRELREC>::= = | > | < | ¬= | >= | <=

<RECOR>::= /RECOR/

<SIMPLE_ASSERTION>::=  /SVASAE1/ <ARITH_EXPRESSION>

         /SVASAE2/ /HEQUAL/ = <ARITH_EXPRESSION> /SVASAE3/

<ARITH_EXPRESSION>::=  /SVAAE1/  [<PLUS_MINUS> /SVAAE2/]

         <OR_EXPRESSION> /SVAAE3/

<OR_EXPRESSION>::= /SVAOR1/ <AND_EXPRESSION> /SVATS2/

  [<RECOR>/SVAOR3/ <OR_EXPRESSION> /SVATS4/]

<AND_EXPRESSION>::= /SVAND1/ <RELATIONAL> /SVATS2/

  [ & /SVAND3/ <AND_EXPRESSION> /SVATS4/]

<RELATIONAL>::= /SVAREL1/ <CATENATION> /SVATS2/

  [<RRELREC> /SVAREL3/ <RELATIONAL> /SVATS4/

  [<RECTO> <ARITH_EXPRESSION> /SVAREL5/ ]

  [ BY <ARITH_EXPRESSION> /SVAREL6/ ]. ]

<CATENATION>::= /SVACAT1/ <TERMS> /SVATS2/

  [<RECCAT> /SVACAT3/ <CATENATION> /SVATS4/]

<RECCAT>::= ||

<RECTO>::= TO

<TERMS>::=  /SVATS1/ <TERM> /SVATS2/ [<PLUS_MINUS>

         /SVATS3/ <TERMS> /SVATS4/]

<TERM>::=  /SVAT1/ <FACTOR> /SVATS2/ [<MOPR>
```

```
          /SVAT3/ <TERM> /SVATS4/]
<FACTOR>::= /SVAF1/ <PRIMARY> [<RECEXP>
          /SVAF3/ <FACTOR> /SVATS4/]
<PRIMARY>::= /WPRIM/ ( <ARITH_EXPRESSION> /WPARAN/ )
          /SVAP2/
  |       <RECINT> /SVAP3/
  |       * /SVAP31/
  |       <FLOAT_DATA> /SVAP4/
  |       <STRING> /SVAP5/[ <LEXBITR>/SVAP51/]
  |       <VARIABLE> /SVAP6/ [ <INDICES> ] /SVAP8/
<LEXBITR>::= /LEXBITR/
<RECINT>::= /RECINT/
<FLOAT_DATA>::= /RECFLOT/
<STRING>::= /LEXCSTR/
<INDICES>::= (<ARITH_EXPRESSION> /SVAP7/
          [,<ARITH_EXPRESSION>/SVAP7/]* /WPARAN/ )
<RECEXP>::= **
<MOPR>::= * | /
<NAME>::= /RECNAME/
<RECFINT>::= /RECFINT/
<DATA_OR_ASSER>::= /WDDS/ <MEDIA_DESCRIPTION>
 | <MOD_STMT>
 | <SFILE_STMT>
 | <FILE_DESC_STMT>
 | <REC_GRP_FLD_INT>
<MOD_STMT>::= <MOD_REC> /SSTMOD/
<SFILE_STMT>::=<S_NAME>[<F_NAME>]/SSTSRC/
<MEDIA_DESCRIPTION>::= <MEDIA_REC> /IMEDIA/[
             <MEDIA_DESC_STMT>] /STMEDIA/

<MEDIA_REC>::= MED | MEDIA
<MEDIA_DESC_STMT>::= (<MEDIADESC> [,<MEDIADESC>]*/WPARAN/)
<MEDIADESC>::= /WMED1/ <TAB_SPEC>
    | <SPACE_SPEC>
    | <DSN_SPEC>
    | <KEY_WORD1> /WMED2/ [<IS>] <SVOPERAND>
    | [<KEY_WORD2> [<IS>] ] <SVOPERAND>
<TAB_SPEC>::= <TAB_REC>[<IS>]/ISPTAB/ <TAB_PARS>/ISPTABE/
<TAB_REC>::= TAB
<TAB_PARS>::= /WTAB/ [(] <RECINT> [ [,]<RECINT>
           /ISPTABI/]* [)]
<SPACE_SPEC>::= <SPACE_REC> [<IS>] <SPACE_PARS>
<SPACE_REC>::= SPACE
<SPACE_PARS>::= /WSPACE/ [(] <RSUNIT> /ISPUNIT/
      [,[<RECINT>/ISPINIT/]  [,[<RECINT>/ISPINC/]
        [,[<RECINT> /ISPDIR/ ] ] ] ]
           [ [,]<RRLSE>/ISPRLSE/] [)]
<RSUNIT>::= TRACKS | TRACK | TRK | CYL | CYLS | BLOCKS
```

```
        | BLK
<RRLSE>::= R | REL | RELEASE | RLSE
<DSN_SPEC>::= <DSN_REC> [<IS>] <DS_NAME>
<DSN_REC>::= DSN | DS | DSNAME | DATASET | FILE_NAME
<DS_NAME>::=/WDSN1/<NAME>/IMDSNF/[./WDSN2/<NAME>/IMDSNN/]*
        [(<MEMBER>/WPARAN/)]
<MEMBER>::= /WMNAME/ <NAME> /IMDMEM/ | [<PLUS_MINUS>
        /IMDMEM/] <RECINT> /IMDMEM1/
<PLUS_MINUS>::= + | -
<KEY_WORD1>::= <RPAGE> | <RLINE> | <RNOTRKS> |
               <RSFILE> | <RDENSTY> | <RINTNAM> |
               <REXTNAM> | <RKEY>
<RPAGE>::= P | LINES_PER_PAGE | PAGE_SIZE
<RLINE>::= LINESIZE | WIDTH | PAGE_WIDTH | LINE_WIDTH
<RNOTRKS>::= TRKS | NO_TRKS | NO_TRACKS | TRACKS
<RSFILE>::= FILE# | START_FILE
<RDENSTY>::= DENSITY | DEN
<RINTNAM>::= IL | INT_NAME | ILABEL
<REXTNAM>::= EL | VOL | VOLUME | VOLSER | ELABEL |
        EXT_NAME
<RKEY>::= KEY
<KEY_WORD2>::= <RBLKNAM> | <RRECNAM> | <RORGN>
             | <RRECFM> | <RUNIT> | <RTLABEL>
             | <RPARTY> | <RCCODE>  | <RDISP>
```

```
<RBLKNAM>::= BS | BLOCKSIZE | BLKSIZE

<RRECNAM>::= RL | LRECL | RECSIZE | RECORDSIZE

<RORGN>::= ORG | ORGANIZATION

<RUNIT>::= UNIT | DEVICE

<RTLABEL>::= LABEL | TAPE_LEBEL | TL

<RPARTY>::= PAR | PARITY

<RCCODE>::= CHAR_CODE | CC | CODE

<RDISP>::= DISP | DISPOSITION

<SVOPERAND>::= <IMORG> | <IURECFM> | <IUNIT> | <IULABEL>

             | <IUPARTY> | <IUCHARC> | <IUDISP> | <IMNOTR>

             | <IMDEN> | <NAME> | <INTEGER>

<IMORG>::= SAM | ISAM | IS | SEQ | SEQUENTIAL |

           INDEXED_SEQUENTIAL | DIRECT | INDEXED |

           REGIONAL | REG | DIR | REG1 | REG2 | REG3

<IURECFM>::= FIXED | VARIABLE | VAR_SPANNED | UNDEFINED

           | FIXED_BLOCKED | F | FB | V | VS | U | VB

           | VARIABLE_BLOCKED

<IUNIT>::= TAPE | 3400-4 | 3400-1 | 3330 | DISK |

           3330-1 | 2314 | 2311 | 2305 | TERMINAL |

           SYSOUT | PRINT | PUNCH | CARD | SYSPRINT |

           SYSIN | OLS | EXPORT | IMPORT | EX | IM

<IULABEL>::= SL | IBM_STD | ANSI_STD | NL | NONE |

          BLP | BYPASS

<IUPARTY>::= ODD | EVEN
```

```
<IUCHARC>::= EBCDIC | BCD | ASCII

<IUDISP>::= OLD | SHR | NEW | MOD

<IMNOTR>::= 7 | 9

<IMDEN>::= 200 | 556 | 800 | 1600

<FILE_DESC_STMT>::=<RFILE> /IMEDIA/ [(/TINPRNT/

        [,<MEDIADESC>]* /WPARAN/)]/STFILE/

<RFILE>::= FILE | REPORT | INPUT | OUTPUT | DATASET

<REC_GRP_FLD_INT>::= <FLD_NAME> /IFIELD/ [<FLD_STMT>]

                                   /STFLD/

<FLD_NAME>::= FIELD | FIE | GROUP | GRP | GRO

           | INTERIM | INT | RECORD | REC | SUB

           | SUBSCRIPT

<FLD_STMT>::= ( /TINPRNT/ [ [,]<FIELD SPEC> ]* /WPARAN/ )

<FIELD_SPEC>::=  ( <OCC_DESC> /WPARAN/ )

    | <FIELD_TYPE> [(/IFPRXL/<LENGTH1>/IFFXDLN/

           [,<RECINT> /IFFXDPR/] /WPARAN/ )

           [VAR /IFSVAR/ ] ]

    | <PIC> [<PICTURE>]

    | <TESTRP> <OCC_DESC>

<TESTRP>::= /TESTRP/

<OCC_DESC>::= <OCC_LIST> [,<OCC_LIST>]*

<OCC_LIST>::=<MIN_OCC> [ : <MAX_OCC>]

<MIN_OCC>::= /WSIZEL/ * /IFOCCS/

    | <RECINT> /IFOCCI/
```

```
     | /IFPRXE/ <VARIABLE> /IFOCCN/
<MAX_OCC>::= /WSIZEU/ * /IFOCCSS/
     |<RECINT> /IFOCCII/
     | /IFPRXE/ <VARIABLE> /IFOCCNN/
<FIELD_TYPE>::= CHA | CHAR | CHARACTER
                | NUM | NUMERIC | BIT | BINARY
                | BIN | DECIMAL | DEC | FIXED
<LENGTH1>::= /WLENTH/ * /IFSFWM2/
     | <RECFINT>
     | <VARIABLE> /IFSFWM1/
<PIC>::= PIC | PICTURE
<PICTURE>::= ' [ <STRING> ] '
```

CHAPTER 6

ASSOCIATIVE MEMORY, DICTIONARY AND PRECEDENCE MATRIX

In this chapter, the organization of the associative memory in MODEL processor is described. The Associative Memory (AM) consists of Storage and retrieval sub-system, Dictionary and the Precedence Matrix. In the following, the above parts of the AM and the routines used to create and access those parts are described. Also, some of the relevant data structures contained in the AM are described. Figure 6.1 gives an overview of the creation and access of Associative Memory, Dictionary, and the Precedence Matrix. In this figure, two-way arrows indicate both creation and access.

6.1 The Storage and Retrieval Subsystem.

6.1.1 Introduction.

During the syntax analysis, each user statement is stored in the associative memory in a coded form. This stored information can be later retrieved or updated. The subsystem for such operations consists of the following routines:

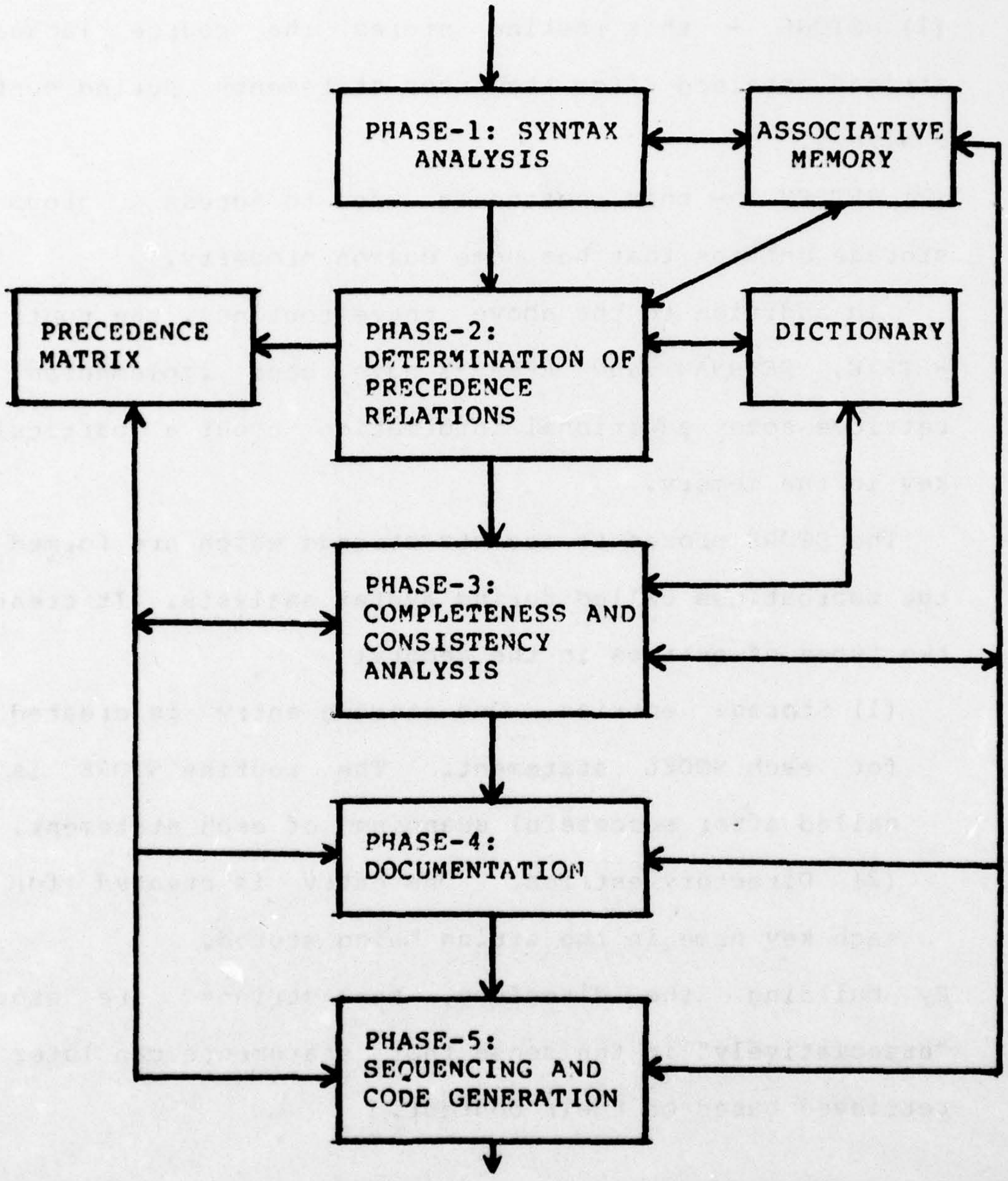

PHASE-1: SYNTAX ANALYSIS
ASSOCIATIVE MEMORY
PRECEDENCE MATRIX
PHASE-2: DETERMINATION OF PRECEDENCE RELATIONS
DICTIONARY
PHASE-3: COMPLETENESS AND CONSISTENCY ANALYSIS
PHASE-4: DOCUMENTATION
PHASE-5: SEQUENCING AND CODE GENERATION

Figure 6.1. Overview of Creation and Access of Associative Memory, Dictionary, and Precedence Matrix.

(1) STORE - this routine stores the source language strings obtained from the user statements during syntax analysis.

(2) RETREVE - this routine is used to access a group of storage entries that has some common property.

In addition to the above three routines, the routines RETR#E, RETRNAM and RETRPRX have been implemented to retrieve some additional information about a particular key in the memory.

The STORE procedure accepts strings which are formed by the subroutines called during syntax analysis. It creates two types of entries in the memory:

(1) Storage entries. One storage entry is created for each MODEL statement. The routine STORE is called after successful scanning of each statement.

(2) Directory entries. One entry is created for each key name in the string being stored.

By building the directory, the strings are stored **"associatively"** in the sense that statements can later be retrieved based on their content.

6.1.2 <u>The Directory and the Storage Structure</u>.

The directory consists of an entry for each key name. Each directory entry points to the first storage entry containing that name. A "linked list" is then maintained from the first storage entry with that key name to other storage entries containing the same key name. A "branch and bound" binary tree structure is chosen for the directory to make the modifications and searching for the key names efficient. Thus, the first key name entered in the directory becomes the root of the directory tree; the next key is entered "above" or "below" it in the tree by lexicographic order; etc.

Each directory entry has the following format:

```
|------------------+---------+---------+---------|
|      key-name    | up-ptr  | down-ptr|  f-ptr  |
|------------------+---------+---------+---------|
```

where

"key-name" is a string of (up to) 10 characters (padded with blanks);

"up-ptr" and "down-ptr" are pointers to other directory entries, whose key names are up or down, respectively, in the lexicographic sense; and

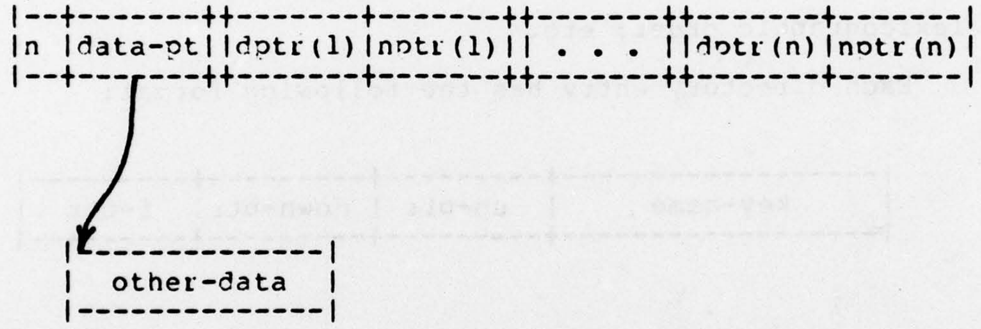
"f-ptr" is a pointer to the first storage entry that contains the key name.

The storage entries (the strings to be stored) consists of two parts:

(1) The key names to be entered in the directory which

include the names the user provided in the MODEL statements for naming data, assertion, etc. These are the names by which we may want to retrieve information later. (2) Auxiliary data from the source language statements. This data is in an encoded form and is not used as basis for retrievals.

The storage entries have the following format:



where

"n" is the number of key names in the storage entry string;

"data-pt" is the pointer to the auxiliary data that contains the coded representation of the source language statement;

"dptr(i)" (i = 1 to n) is the directory entry pointer to the ith key name; and

"nptr(i)" (i = 1 to n) is the pointer to the next storage entry which contains the same key name.

In general, the name represented by the second key name in the storage entry (dptr(2)) is the data name (or the assertion name) that the storage entry represents.

6.1.3 Key Names.

The names supplied by the user for naming data, assertions, etc., are prefixed by a two character prefix to obtain the key name for the corresponding name. The prefix identifies the type of name that is being stored. For example, the prefix is '$G' for data names that represent groups, '$A' for assertion names, '#E' for **"exist"** names, and so on. The following table gives the allowed prefixes and the description of what they represent.

TABLE 6.1

PREFIX	NAME TYPE
$A	Assertion name
#A	Qualified Ass. name
$B	Subscript name
$C	Record name
$D	Media name
#E	Exist name
$F	File name
$G	Group name
$I	Interim name
#K	Key name
$L	Field name
$M	Module name
$P	Parent name
#Q	Qualified name
$S	Source var. name
$T	Target var. name

In addition, the first key name in any storage entry is a special two character name which identifies the type of statement the storage entry represents. The different statement types are given in the following table:

TABLE 6.2

First key name in the storage entry	Statement type
$A	Assertion
$B	Subscript statement
$C	Record statement
$D	Media statement
$F	File statement
$G	Group statement
$I	Interim statement
$L	Field statement
$M	Module statement
$S	Source file statement
$T	Target file statement

If a data name supplied by the user is not longer than 8 characters, then the data name is directly stored (after prefixing with an appropriate prefix) in the directory. If the name is longer than 8 charaters, then the directory name contains a "coded name" which is 4 characters long. This coded name along with the two character prefix is stored in the directory. The names that are longer than 8 characters are saved in a tree structure, and the coded name is actually the index of that name in the tree structure. The maximum allowed length for any data name is 31.

6.1.4 The Data Area of Storage Entries.

Each storage entry contains a pointer (data-pt) to the "data area" (auxiliary data) which contains a coded information of the MODEL statement that the storage entry represents. Most of the information that is contained in the data area is filled in during the syntax analysis. Some of the entries are filled in during the creation of the "dictionary" which is described in the later sections. The data area consists of two parts: (1) "common data area" and (2) "extended data area" in the following format:

```
|-----------------------+-----------------------|
|    common data area   |  extended data area   |
|-----------------------+-----------------------|
```

6.1.4.1 The Common Data Area.

The common data area is common to all types of statements. This contains 5 entries in the following format:

```
|------------------|
|   comment-ptr    |
|------------------|
|    user-line#    |
|------------------|
|    proc-line#    |
|------------------|
|    data-list     |
|------------------|
|     reserved     |
|------------------|
```

where

"comment-ptr" is pointer to a string that contains the comment associated with the statement.

"user-line#" is the input line number in which the statement begins.

"proc-line#" is the statement number supplied by the system.

"data-list" is a pointer to a block of data which is filled during the creation of dictionary.

The pointer, data-list, points to the data block which has the following format:

```
|-------------------|
|    parent-ptr     |
|-------------------|
| index1 | index2   |
|-------------------|
|   status-bits     |
|-------------------|
|     dtr-ptr       |
|-------------------|
```

where

"parent-ptr" is the storage entry pointer of the parent data name for this data name.

"index1" is the dictionary index for this data name.

"index2" is a duplicate dictionary index of the same same data name (which is used for the data names in an update file).

"status-bits" is a set of bits that indicates the type of data name (source or target). There are seven bits in this set. The significance of these bits is given by the following table:

TABLE 6.3

Status bit	Significance
SRC_BY_DDS	Defined as descendant of a source file
TAR_By_DDS	Defined as descendant of a target file
SRC_BY_ASS	Used as source of an assertion
TAR_BY_ASS	Used as target of an assertion
SRC_BY_RES	Resolved as source variable
TAR_BY_RES	Resolved as target varaible
INT_BY_RES	Resolved as interim variable
UPD_BY_RES	Resolved as Update variable

"dtr-ptr" is a pointer to a linked list that contains the storage entry pointers of all the immediate descendants (daughters) of this data name. This linked list contains entries in the following format:

```
|-------------+--------------|
| ste-pointer |   sis-ptr    |
|-------------+--------------|
```

where

"ste-pointer" is a pointer to the storage entry of a daughter name; and

"sis-ptr" is a pointer to a similar structure above, which contains the storage entry pointer of the next daughter name.

6.1.4.2 The Extended Data Area.

The Extended data area contains some additional information about the statement. There are three types of extended data areas. These three types are used in the following groups of statements:

(1) assertions;

(2) Media and File statements; and

(3) Record, Group, Field, Interim and Subscript statements.

The data area of the Module name statements, Source File statements, Target File statements and Refer statements consists of only the common data area described above. In the following, the three types of extended data areas are described in more detail.

6.1.4.3 Extended Data Area for Assertions.

The format of the extended data area for assertions is given in the following:

```
|-------------------|
|      vl-ptr       |
|-------------------|
|      ass-ptr      |
|-------------------|
|     nass-ptr      |
|-------------------|
|      sv-ptr       |
|-------------------|
|      tv-ptr       |
|-------------------|
|      fn-ptr       |
|-------------------|
|     init-ptr      |
|-------------------|
|     test-ptr      |
|-------------------|
|      sol-ptr      |
|-------------------|
```

where

"vl-ptr" is a pointer to a list of variables used in the assertion. This pointer is obtained during the creation of the dictionary.

"ass-ptr" is a pointer to the derivation tree of the assertion. This derivation tree is generated during the syntax analysis (see also the description of DT in section 6.4).

"nass-ptr" is a pointer to the data area or storage entry of another assertion if the two assertions belong to a set of simultaneous equations.

"sv-ptr" and "tv-ptr" are pointers to the source and target variable list, respectively, for this assertion. This list is specified by the user.

"fn-ptr" is a pointer to a list of function names used in the assertion.

"init-ptr" is a pointer to a list of initial values for the variables used in the assertion.

"test-ptr" is a pointer to a list of conditions associated with each variable for termination of the iteration in the solution of the simultaneous equations.

"sol-ptr" is a pointer to a list of solution methods to be used (in the case where numerical analysis procedures are to be used, such as, with the solution of simultaneous equations).

All the above pointers, except, ass-ptr and nass-ptr point to a dynamic list of entries having the following format:

```
|---+---------+---------+---------|
| n | ptr(l)  |  . . .  | ptr(n)  |
|---+---------+---------+---------|
```

where

"n" is the number of elements in the list; and

"ptr(i)" (i = 1 to n) is a pointer to the ith element in the list. This pointer points to the derivation tree of an "arithmatic expression" in the case of sv-ptr, tv-ptr, fn-ptr, and sol-ptr. It points to the derivation tree of a "simple assertion" in the case of test-ptr and init-ptr. It points to a structure "var-list" (described later) in the case of vl-ptr.

The pointer, ass-ptr, in the extended data area above, points to the derivation tree of the assertion itself; whereas, the pointer, nass-ptr, points to a structure which has the following format:

```
|------+-------------+-------------|
| code |  first-ptr  |  next-ptr   |
|------+-------------+-------------|
```

where

"code" is 'D', 'S' or blank and represents the type of pointer specified by next-ptr.

"first-ptr" is a storage entry pointer of the first assertion in a set of simultaneous equations.

"next-ptr" is a pointer to the data area (if code = 'D') of another assertion having exactly the same name, or it points to the storage entry (if the code = 'S') of another assertion which belong to a set of simultaneous equations. If the code is blank, then the assertion represented by the current data-area is the last assertion in the set.

AD-A063 900 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN--ETC(U)
SEP 78 S K SHASTRY N00014-76-C-0416
UNCLASSIFIED 78-01 NL

3 OF 6
AD
A063900

In general, one storage entry exists for each assertion name. If there exists two or more assertions with the same assertion name, then those assertions are linked together (with code = 'D') as described above. Two or more assertions having different names can also form a simultaneous set, if either, a simple name in one assertion name is contained in another assertion name (for example, in the assertion names 'A.B' and 'A.C', the simple name 'A' is contained in both), or those assertions that were resolved to be a simultaneous set during the cycle analysis. In such cases, the data areas of the assertions that form the simultaneous set are linked together as described above, with code = 'S'.

6.1.4.4 <u>Extended Data Area for Media and File Statements</u>.

The format of the extended data area for Media and File statements is given below:

BLOCKSIZE	RECORDSIZE	PAGESIZE	LINESIZE	SPACE
(2)	(2)	(2)	(2)	(8)

START-FILE	TAB	UNUSED	ORG	RECFM	RLSE	DENSITY	TRKS
(2)	(20)	(6)	(1)	(1)	(1)	(1)	(1)

TLABEL	PARITY	CHAR-CODE	INT-EXT-NAME	UNIT	DISP
(1)	(1)	(1)	(4)	(1)	(1)

MEMBER NAME	DATA SET NAME
(8)	(44)

Each entry in the above list represent one of the parameter associated with the statement. The number below each entry (enclosed in parentheses) is the length (in bytes) of that entry. Each parameter of a media or file statement is summarized in table 4.1 of Chapter 4.

6.1.4.5 Extended Data Area for Other Data Statements.

The format of the extended data area for record, field, group, interim and subscript statements is given below:

FIELD-LENGTH	PICTURE-ATTR	PRECISION	FIELD-TYPE	VARYING
(4)	(4)	(2)	(1)	(1/8)

DIMENSION	BITS	LB(1)	UB(1)	LB(2)	UB(2)	...
(2)	(2)	(4)	(4)	(4)	(4)	

In the above list, FIELD-LENGTH specifies the length of field, PICTURE-ATTR represents a pointer to the Picture attribute associated with the field, PRECISION specifies the precision associated with the decimal data, FIELD-TYPE specifies the type of field ('B' for binary, 'C' for character, 'N' for numeric, 'D' for decimal, and 'T' for bit), DIMENSION specifies the dimension of the data name, LB(i) and UB(i) specify lower bound and upper bound associated with the size of the i th dimension. If a bound of a dimension is a data name, or '*' then the corresponding bit in BITS is set. Also, if the field length is varying, then the bit VARYING is set. Note that the first four entries are allowed for only the field and interim data names. Also, in the case of subscript statement, UB(1), UB(2) and UB(3) specify the lower-bound, upper-bound and increment, respectively, associated with the subscript name; and UB(4) specifies the dimension of the parent data name of the subscript with which it is associated.

6.1.5 Creating and Accessing the Associative Memory.

This section describes the different routines used to create and access the directory and storage entries of the associative entries. The following routines are used:

(1). STORE
(2). RETREVE
(3). RETR#E
(4). RETRNAM
(5). RETRPRX

6.1.5.1 <u>The STORE Procedure</u> (<u>Store Statements</u>).

The STORE(S,D) procedure has two parameters, S and D. S is the string containing the key names which are to be stored and to be entered in the directory. D is the pointer to the previously built auxiliary data from the source string. The latter is an encoded form of non-key source language information.

The detailed algorithm for the STORE procedure is given in RIN76.

6.1.5.2 <u>The RETREVE Procedure</u> (<u>Retreve Storage Entries</u>).

The RETREVE(E,D,S,N,P) is the procedure for retrieving desired storage entries, by searching through the associative memory entries. It has 5 parameters. RETREVE finds all the storage entries in which the given key name or expression of key names, E, appears and furthermore checks whether the first characters of data associated with the storage entries match the string D. That is, RETREVE finds all the storage entries with key satisfying the logical expression E and the other data D. RETREVE starts its search at directory entry S, normally the root of the directory, and it returns a list of pointers P, to those entries which satisfy the request by the calling program. The number of storage entries satisfying the request is returned in N.

The logical expression E used to retrieve strings can be any boolean expression involving the keynames in disjunctive normal form, where the first key in each term is non-negated. For example, the following call:

```
  CALL RETREVE('$G        &$GITEMS    ','',START,N,P)
```

will return in N, all the storage entry pointers which contain both the key names '$G' and '$GITEMS'. Number of such storage entries is returned in P.

The detailed algorithm for the RETREVE procedure can be found in RIN76.

6.1.5.3 <u>The Function RETR#E (Retreve # of Entries)</u>.

The function RETR#E(S) returns the number of storage entries in which the key name S occurs. Usually, this routine is used in conjunction with the RETREVE procedure to obtain the size of the array of pointers (fourth argument) in which the required storage entry pointers are to be returned. The following sequence of PL/I statements illustrates the use of the function RETR#E:

```
        DCL N BIN FIXED;
        DCL T(N) PTR CTL;
          N = RETR#E('$GITEMS');
          ALLOCATE T;
          CALL RETREVE('$G  &$GITEMS','',START,T,#);
           ...
          FREE T;
```

Note that the argument S of RETR#E can only be a single key name.

6.1.5.4 <u>The RETRNAM Procedure</u> (<u>Retreve by Name</u>).

The RETRNAM(S,D,N) returns in D, all the key names in the directory which match with the string S, except for the first two character prefix. The number of such names is returned in N. For example, if the directory contains the key names, '$GX', '$GY' and '$PX', then the following call:

```
CALL RETRNAM('$GX',D,N)
```

will return in D, the two names, '$GX' and '$PX'; and "2" is returned in N.

6.1.5.5 <u>The RETRPRX Procedure</u> (<u>Retreve by Prefix</u>).

The RETRPRX(S,D,N) returns in D, all the key names in the directory, the prefix of which is contained in the string S. The number of names found is returned in N. In the example of section 6.1.5.4, the following call:

```
CALL RETRPRX('$G',D,N)
```

will return in D, the two names, '$GX' and '$GY'; and "2" is returned in N.

6.2 The Dictionary.

The dictionary contains entries for the data names and assertion names supplied by the user. It is created during the semantic analysis of statements stored in the AM which is described in chapter 8. In general, an assertion name can have only one dictionary entry, whereas, a data name can have more than one entry if that data name is used with different subscripts. A dictionary entry has the following format:

steptr	sub-ptr	sub#	nindex	adjmptr	wptr

where

"**steptr**" is a storage entry pointer of the data name that this dictionary entry represents;

"**sub-ptr**" is a pointer to the subscript list, if any, asoociated with that data name;

"**sub#**" is the dimensionality of the data name;

"**nindex**" is the next dictionary index of the same data name, but with different subscript;

"**adjmptr**" is a pointer to a list that contains the "precedence matrix" entries for this dictionary entry; and "wptr" is a pointer to the list of structures that define the map vector associated with each type-8 edge that emanates from this node.

Note that a data name can have more than one dictionary entry in the following situations:

(1) The data name is a descendant of an update file; or

(2) The data name is an array.

In the latter case, each use of the data name with different subscript has an entry in the dictionary.

For example, if MATRIX is an array of size 10x10, and it is used in three forms: MATRIX(1,8), MATRIX, and MATRIX(5,*), then 3 dictionary entries will be created for the three different representations.

6.2.1 Description of Subscripts.

A subscript associated with a data name (pointed by sub-ptr in the dictionary) is described by the following PL/I structure:

```
DCL 1 VAR_LIST BASED(VARLISTP),
    2 SRC_OR_TAR CHAR(1),
       /*'S' - IF THE VAR IS ON THE RHS OF ASSERTION
         OR IN THE <IF CLAUSE> OF THE ASSERTION,
         'T' - IF ON THE LHS OF ASSERTION,*/
    2 ELSE_BIT BIT(1), /*THIS BIT IS SET IF THE VARIABLE
          IS IN THE ELSE CALUSE OF THE ASSERTION*/
    2 RED_FN BIT(1), /*SET IF THIS NAME IS ARGUMENT OF
                        A REDUCTION FUNCTION*/
    2 #OFI BIN FIXED, /*NUMBER OF SUBSCRIPTS*/
    2 LINDEXO BIN FIXED, /*DICTIONARY INDEX*/
    2 VAR_NAM_PTR PTR, /*POINTER TO VARIABLE NAME*/
    2 SUBSCRIPTS(N REFER(VAR_LIST.#OFI)),
       3 NAM_OR_INT BIT(4), /*'0'B IF INTEGER,
                     '1'B OTHERWISE.*/
       3 FCN_BIT BIT(1), /*SET IF FUNCTION*/
       3 CONST_BIT BIT(1), /*SET IF THE SUBSCRIPT
                      IS A CONSTANT*/
       3 VAR_SUB_PTR PTR, /*POINTER TO SUBSCRIPT NAME*/
       3 LB BIN FIXED(31,0), /*LOWER BOUND*/
       3 UB BIN FIXED(31,0), /*UPPER BOUND*/
       3 INC BIN FIXED(31,0), /*INCREMENT*/
       3 OFF BIN FIXED(31,0); /* OFFSET*/
         /*I = LB TO UB BY INC A(I+OFF)*/
```

6.2.2 Description of Map Vector.

The pointer, MPTR, in each dictionary entry points to the following structure, which describes the map vectors associated with all the type-8 successors of the node.

```
DCL 1 WEIGHTV_LIST BASED(WVPTR),
        2 WDIM BIN FIXED, /*DIMENSION*/
        2 #TYPE8 BIN FIXED, /*# OF TYPE 8 EDGES*/
        2 W_V (N REFER(#TYPE8)), /*WEIGHT VECTORS*/
          3 X0 BIN FIXED, /*LOWER BOUND*/
          3 XM BIN FIXED, /*UPPER BOUND*/
          3 CX BIN FIXED, /*INCREMENT*/
          3 Y0 BIN FIXED, /*LOWER BOUND*/
          3 YM BIN FIXED, /*UPPER BOUND*/
          3 CY BIN FIXED; /*INCREMENT*/
```

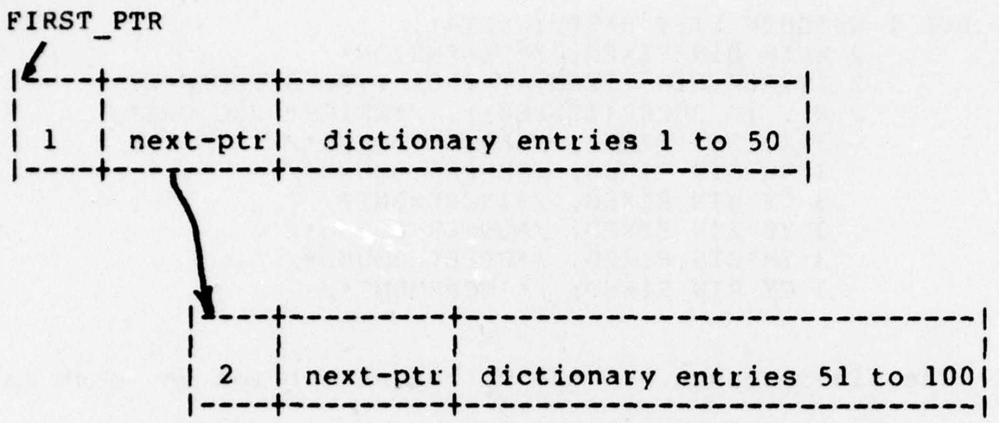
The significance of the different entries in each map vector is described in chapter 8.

6.2.3 <u>Creating and Accessing the Dictionary.</u>

In this section, the different routines that are used to create and access the dictionary are described briefly. The following routines are used:

(1). ACRD0
(2). ACRD1
(3). ACRDR
(4). ACRDC
(5). ACRDRA
(6). ACRDRS
(7). ACRDFP

Even though, the dictionary was described in the earlier section as an entry for each data name, it is actually organized in a group, each group containing 50 entries as follows:



... and so on.

In order to access i th dictionary entry, it is necessary to search thru the list of stacks for the j th stack and then access the k th entry in that stack; where,

$$j = 1 + i / 50$$

and $$k = 1 + MOD(i-1,50)$$

6.2.3.1 <u>The ACRDO Procedure</u>.

The ACRDO(S) procedure creates a dictionary entry for the data name with the storage entry pointer S. It also creates a data area which is linked to data-list of the common data area of the storage entry (see section 6.1.4.1). This list is used to contain pointers that link the storage entry with its parent and daughter names. This routine is usually called to add an assertion name (or a data name in a data description statement) to the dictionary.

6.2.3.2 The ACRD1 Procedure.

The ACRD1(S,I,RI) procedure creates a dictionary entry for the data name with the storage entry pointer S. Also, I specifies the dimensionality of the data name, and if it is zero, then the use of the data name did not have any subscript. The subscript description of the data name is given by the pointer to a structure VAR_LIST (described later). This pointer is given by the external pointer VARLISTP. The index of the dictionary entry created is returned in RI.

The function of this routine is to search the dictionary for the same data name and the subscript. If an entry is found, then no entry is created. Otherwise, an additional entry is created by properly linking this entry to the previous entry of the same name.

6.2.3.3 The ACRDR Procedure (Return Storage Entry Ptr).

The ACRDR(I,PTR) procedure returns the storage entry pointer of the Ith dictionary entry in the second argument, PTR. The pointer to subscript information associated with that dictionary entry is returned in the pointer VARLISTP (which is an external pointer).

6.2.3.4 The ACRDC Procedure (Duplicate Dict. Entry).

The ACRDC(PTR) procedure creates a duplicate dictionary entry for the storage entry given by PTR. This routine is used to create duplicate entries for the descendants of an update file.

6.2.3.5 The ACRDRA Procedure (Return PM Entry List Ptr.).

The ACRDRA(I,PTR) procedure returns in the second argument the pointer to the precedence matrix list, adjmptr, for the Ith dictionary entry.

6.2.3.6 The ACRDRS Procedure (Set PM Entry List Ptr.).

The routine ACRDRS(I,PTR) is used to set the Ith adjmptr with the pointer given in the second argument, PTR.

6.2.3.7 The ACRDFP Procedure (Return First Stack Pointer).

The function ACRDFP returns the pointer to the first stack of the dictionary entries (FIRST_PTR in the figure described in the beginning of section 6.2.4).

6.3 The Precedence Matrix.

The Precedence Matrix (Weighted Adjacency Matrix) is a non-pictorial representation of the directed graph that represents the precedence relationships between the data names and the assertion names in the MODEL specification. It is a matrix of size nxn, where, n is the number of dictionary entries used in the MODEL specification. If any element (i,j) of the precedence matrix is nonzero then the j th dictionary entry is said to be the successor of the i th dictionary entry. The (i,j) th element in the matrix represents the type of relation between the two dictionary entries. The allowed types of relations are described in section 4.12. This section describes the organization of the precedence matrix.

To conserve space for saving the nxn matrix and to minimize time required to search an entry in the precedence matrix, the matrix is not represented in the form of an array, but with n different lists. The i th list contains a list of all the entries in the i th row and the i th column of the precedence matrix. The address of this list can be obtained from the i th dictionary entry (the routine ACRDRA can be used for this purpose. See section 6.2.4). The format of the precedence matrix entry list corresponding to the k th dictionary entry is given in the following:

```
|-----+-----+-----++------+------++------+------++-----+|
|alloc|rused|cused||col(1)|cvalue||row(1)|rvalue||  ...||
|     |     |     ||      |  (1) ||      |  (1) ||     ||
|-----+-----+-----++------+------++------+------++-----+|

        |+-----++------+------++------+------++-----+|
        ||  ...||col(i)|cvalue||row(i)|rvalue||  ...||
        ||     ||      | (i)  ||      | (i)  ||     ||
        |+-----++------+------++------+------++-----+|
```

where

"alloc" is the maximum number of column or row entries in this list (that is, alloc >= *MAX(rused,cused)*).

"rused" is the number of entries in the k th row.

"cused" is the number of entries in the k th column.

"col(i)" and "cvalue(i)" (i = 1 to rused) is the column number and the corresponding matrix entry (in k th row and col(i) th column) respectively.

"row(i)" and "rvalue(i)" (i = 1 ro cused) is the row number and the corresponding matrix entry (in row(i) th row and k th column) respectively.

As an example, if the entries (5,100), (5,8) and (10,5) of the precedence matrix are 3, 4, and 1 respectively, then the 5th precedence matrix entry list will look as follows:

2	2	1	100	3	10	1	8	4	-	-

The 8 th precedence matrix entry will look as follows:

2	0	1	-	-	5	4	-	-	-	-

The 10th precedence matrix entry will look as follows:

2	1	0	5	1	-	-	-	-	-	-

And finally, the 100 th precedence matrix entry will look as follows:

2	0	1	-	-	5	3	-	-	-	-

6.3.1 Creating and Accessing the Precedence Matrix.

This section describes the routines used to create and access the precedence matrix. The routines used are:

(1) ADJMRS

(2) ADJMCS

(3) ADJMSET

(4) ADJM

6.3.1.1 <u>The ADJMRS Procedure (Set a Row of PM)</u>.

The ADJMRS(I,V) procedure updates the precedence matrix entry list of the Ith dictionary entry with the value specified by the second argument, V. V is a matrix of size 2xN. The first row of V specifies the column numbers and the second row specifies the values that the corresponding columns in the first row must have. For example, if V is given by the following matrix:

```
10   5   2
 3   6   1
```

then the following call:

```
CALL ADJMRS(12,V)
```

will set the values 3, 6, and 1 to the locations (12,10), (12,5) and (12,2) respectively.

6.3.1.2 <u>The ADJMCS Procedure (Set a Column of PM)</u>.

The ADJMCS(I,V) PROCEDURE is similar in function to the procedure ADJMRS. In this case I represents the column number (I represented the row number in the case of ADJMRS). In other words, ADJMCS is used to set one or more entries of the Ith column of the precedence matrix.

6.3.1.3 The ADJMSET Procedure (Set an Entry of PM).

The ADJMSET(I,J,K) procedure is used to set the value K to the entry (I,J) of the Precedence matrix.

6.3.1.4 The ADJM Procedure (Return an entry of PM).

The Function ADJM(I,J) returns the entry (value) in the Ith row and the Jth column of the precedence matrix. If the entry (I,J) is undefined, then '0' is returned. Also, If I = 0, then it returns the number of nonzero entries in the Jth column, and if J = 0, then it returns the number of nonzero entries in the Ith row.

6.4 The Derivation Tree of an Assertion.

For ease of analyzing an assertion, the derivation tree (DT) of each assertion is saved in the data area of the assertion. The DT is a list of entries that represent each production in the EBNF for the assertion. The root of the DT is saved in ASS_PTR of the extended data area (see section 6.1.4.3). The different entries used in the DT of an assertion is given in the following sections. Some examples of DTs are presented in the section 6.4.10.

6.4.1 The Assertion Entry.

The pointer ASS_PTR, in the extended data area of an assertion points to the root of the dirivation tree of the assertion. This root entry has the following format:

```
|----------+----------+----------+----------+----------|
|  if-ptr  |as-ptr(1) |sas-ptr(1)|as-ptr(2) |sas-ptr(2)|
|----------+----------+----------+----------+----------|
```

where

"if-ptr" is a pointer to the "if-clause" entry. If no "if-clause" exists then the pointers: if-ptr, as-ptr(1), as-ptr(2), and sas-ptr(2) are all null.

"as-ptr(1)" is pointer to the next assertion entry (similar to the above entry), if the assertion contains nested "if-clause"s.

"sas-ptr(1)" is a pointer to the "simple assertion" entry.

"as-ptr(2)" is pointer to the next "assertion" entry (similar to the above entry) if the assertion contains nested "if-clause"s in the "else" part of the assertion.

"sas-ptr(2)" is a pointer to the "simple-assertion" entry in the "else" part of the assertion.

For example, if the assertion is given by:

```
IF A=B THEN IF A=C THEN X=Y ELSE X=Z
```

then if-ptr points to the DT of the boolean expression

"A=B", as-ptr(1) points to the DT that represents the assertion "IF A=C THEN X=Y ELSE X=Z", and the pointers sas-ptr(1), as-ptr(2) and sas-ptr(2) are all null.

As another example, for the assertion:

IF A=C THEN X=Y ELSE X=Z

if-ptr points to the DT of the boolean expression "A=C", sas-ptr(1) points to the DT of the simple assertion "X=Y", and sas-ptr(2) points to the DT of the simple assertion "X=Z". The pointers as-ptr(1) and as-ptr(2) are both null.

Some more examples of the derivation trees are given in the section 6.4.10.

6.4.2 <u>The Simple Assertion Entry</u>.

The simple assertion entry has the following format:

```
|---------+---------|
| lhs-ptr | rhs-ptr |
|---------+---------|
```

where

"lhs-ptr" is a pointer to the DT of the arithmetic expression in the LHS of the assertion,

and "rhs-ptr" is a pointer to the DT of the arithmetic expression in the RHS of the assertion.

6.4.3 <u>The</u> <u>Arithmetic</u> <u>Expression</u> <u>Entry</u>.

The arithmetic expression entry has the following format:

```
|-----+-----+-----------|
|code |opr  |terms-ptr  |
|-----+-----+-----------|
```

where

"code" indicates the type of DT pointed by terms-ptr,

"opr" is a monadic operator, if any,

and "terms-ptr" points to one of the 9 entries depending on the value of code. The different types of DTs that terms-ptr points to, is given in the following table.

TABLE 6.4

code	DT pointed by terms-ptr
A	Arithmetic expression
R	Expression using relational operators
|	Expressions using '|'
&	Expressions using '&'
C	Expressions using '||'
S	Expressions using '+' or '-'
T	Expressions using '*' or '/'
F	Expressions using '**'
I	Integer
M	Misc-data
V	Variable name

6.4.4 <u>The</u> <u>Diadic</u> <u>Expression</u> <u>Entry</u>.

The expressions using the diadic operators '|', '&', '||', '+', '-', '*', '/', '**' and the relational operators have the same format and is given by:

left-ptr	lcode	opr	rcode	right-ptr

where

"left-ptr" points to one of the 9 entries depending on the value of code (as in the case of arithmetic expression entry),

"lcode" indicates the type of pointer represented by left-ptr (has the same meaning as the code of arithmetic expression entry),

"rcode" indicates the type of pointer represented by "right-ptr" (has the same meaning as the code of arithmetic expression entry),

"opr" represents one of the diadic operators: |, &, ||, +, -, *, /, **, <, >, <=, >=, =, =,

"right-ptr" is a pointer to RHS of the diadic expression.

If rcode = 'X' then right-ptr points to an extended area which is used to save the upper bound and increment associated with the for expression. This extended area has the following format:

```
|------+----------+----------+----------|
|xcode |rptr      |toptr     |byptr     |
|------+----------+----------+----------|
```

where

"xcode" indicates the type of pointer represented by rptr,

"rptr" is a pointer to the RHS of the diadic expression,

"toptr" is a pointer to the arithmetic expression that
specifies the upper bound, and

"byptr" is a pointer to the arithmetic expression that
specifies the increment associated with the subscript
name.

6.4.5 <u>The Integer Entry</u>.

The entry for "integer" represents an integer constant and has the following format:

```
|---------|
|value    |
|---------|
```

where "value" is the integer constant itself.

6.4.6 <u>The Misc-data Entry</u>.

The entry for "Misc-data" represents bit-strings, character strings and floating point data constants and has the following format:

```
|------+------+---------------|
|type  |length|data           |
|------+------+---------------|
```

where

"type" specifies the type of data represented by data,

"length" is the length of the data that follows,

and "data" is BIT (if type = 'B') or character representation (if type = 'C'), or floating point representation (if type = 'F') of the data.

6.4.7 <u>The Entry for Variable Names</u>.

The entry for variable names has the following format:

```
|------+----------------------|
|length|variable name         |
|------+----------------------|
```

where

"length" is the length of the variable name that follows,

and "variable name" is the name of the variable itself.

6.4.8 <u>The Variable Entry</u>.

The variable entry represents a subscripted variable name and has the following format:

```
|---------+----+----------+----------+----------|
| nam-ptr | n  |  ae(1)   |  . . .   |  ae(n)   |
|---------+----+----------+----------+----------|
```

where

"nam-ptr" is a pointer to the variable name entry (see section 6.4.7),

"n" is the number of subscripts associated with the variable name,

and "ae(i)" (i = 1 to n) is a pointer to the arithmetic expression entry that represents the i th subscript.

6.4.9 The Boolean Expression Entry.

The boolean expression entry has the same format as the arithmetic expression entry described in section 6.4.3. The LHS or RHS of diadic expressions that use '|' or '&' as operators, specify individual for expression or boolean expression. In the case of for-expressions, the diadic expression entry can have extended information (rcpde = 'X') as explained in section 6.4.4.

6.4.10 Some Examples of Derivation Trees.

In the following, the derivation trees for some assertions are given. The assertions used are:

EXAMPLE1: A = B + C

EXAMPLE2: IF A=B THEN C=D(I) ELSE C = E

EXAMPLE3: FOR I = 1 TO 50 BY 2 A(I*10+2) =B(I)

The derivation trees for the above examples are given in Figures 6.2, 6.3 and 6.4 respectively.

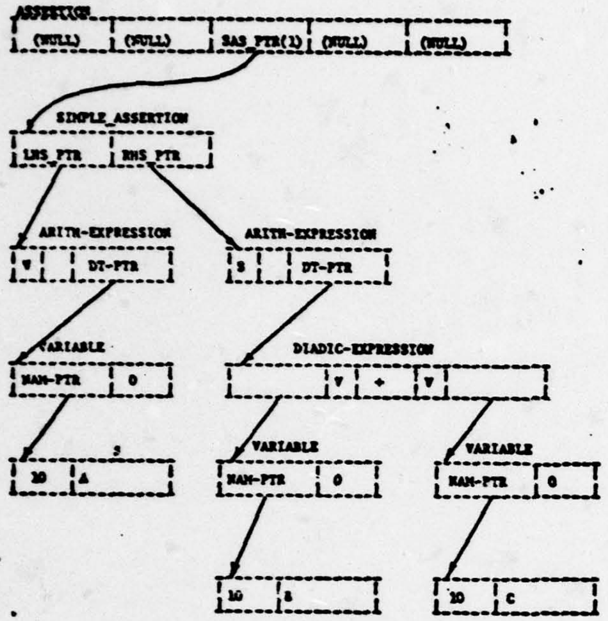


Figure 6.2.
Derivation tree for the assertion:
A = B + C

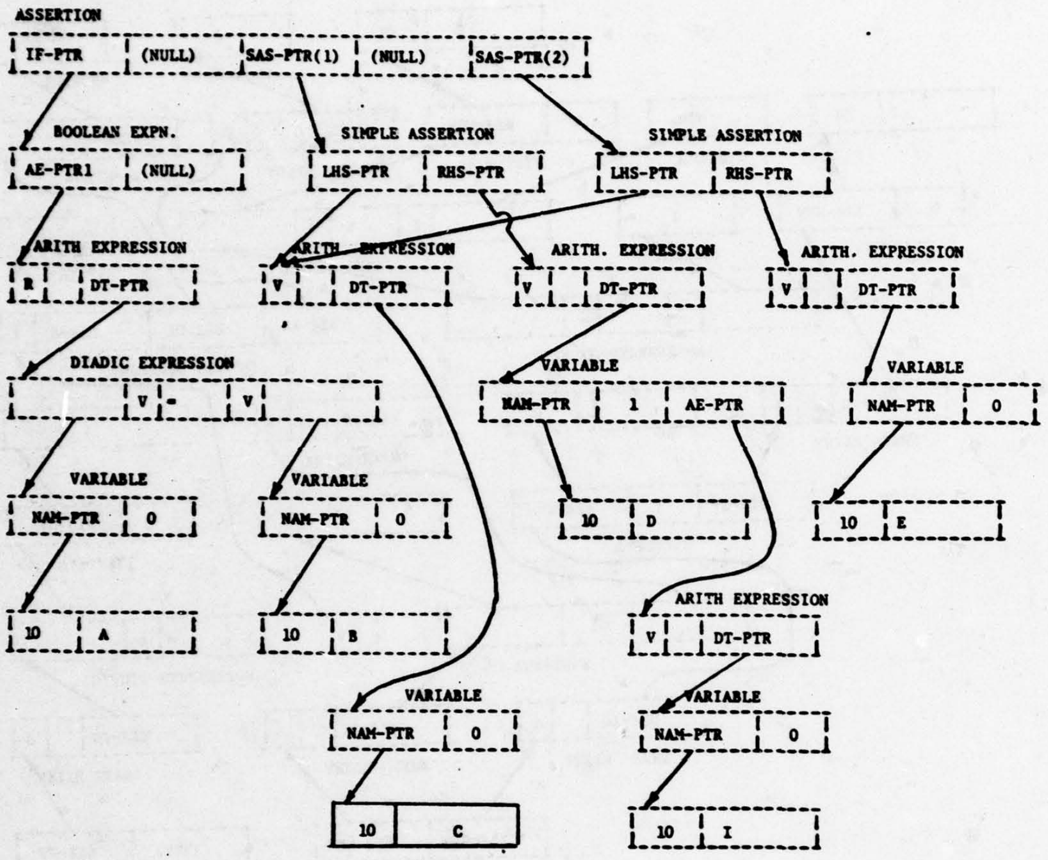


Figure 6.3.
Derivation Tree for the Assertion:
IF A = B THEN C=D(I) ELSE C = E

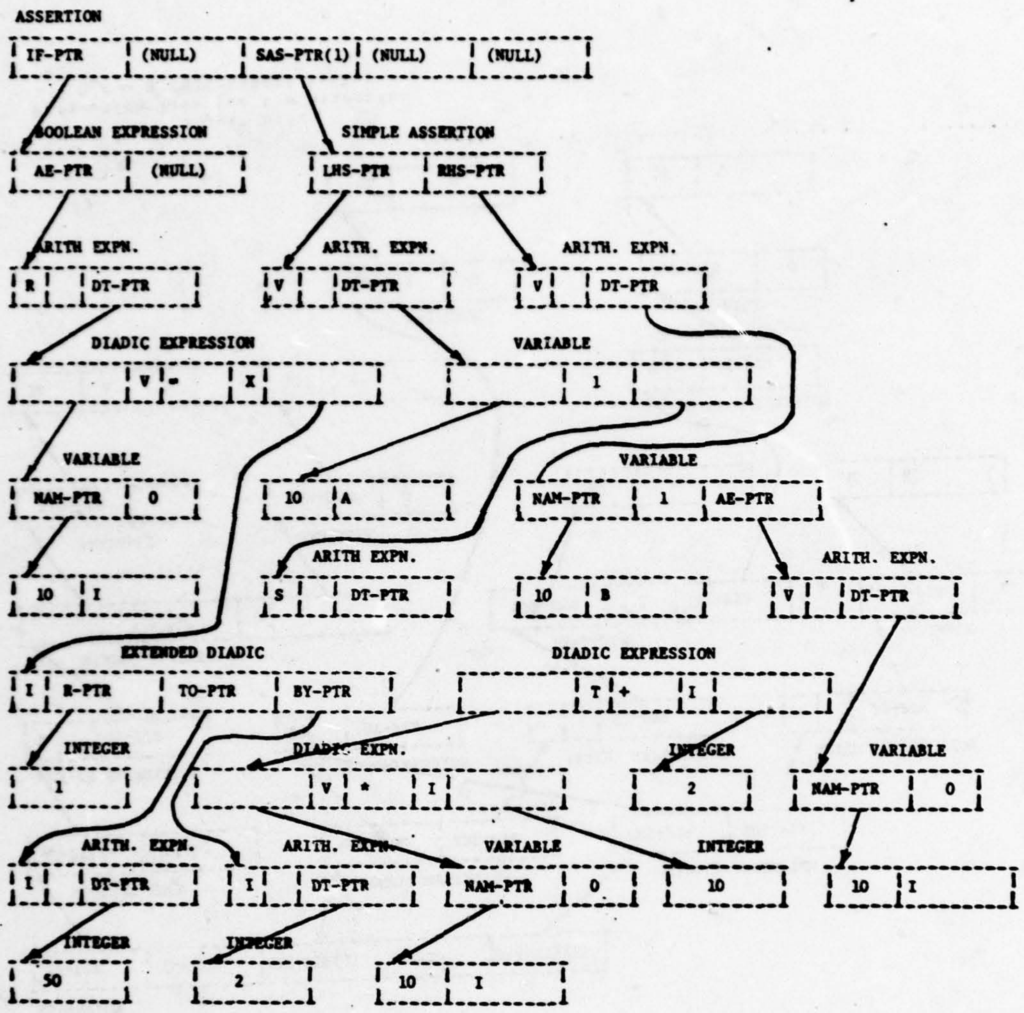


Figure 6.4.
Derivation Tree for the Assertion:
FOR I = 1 TO 50 BY 2 A(I*10+2) = B(I)

6.5 The Precedence Relations.

In this section, the allowed precedence relations between the data names and the assertion names (the different entries of the precedence matrix) are described: Essentially, there are three groups of relationships.

(1). Hierarchical Relationship - This is a relationship between a data name and its descendant (or its parent).

(2). Value Dependency Relationship - This is a relationship between a data name and an assertion name.

(3). Pointing Relationship - This is a relationship between an exist, length or pointer name and a data name.

The entries in the precedence matrix can contain the 9 types of relationships, which fall into one of the above categories. These types of relations are described in the following:

Typel: This type is a hierarchical relationship between the source data names and its descendants.

Type2: This type is also a hierarchical relationship and is between the target data names and their parent.

Type3: This type is a value dependency relationship between the source variables of an assertion and the assertion itself.

Type4: This type is also a value dependency relationship, and is between an assertion and target variables used in that assertion.

Type5: This type is a pointing relationship between a "length name", L, and a data name the length of which is given by L.

Type6: This type is also a pointing relationship, and is between an "exist name", E, and a data name, the size of which depends on E.

Type7: This type is also a pointing relationship and is between a "pointer name", P, and a repeating data name, the selection of a particular member of which is determined by P.

Type8: This type is a hierarchical relationship between entries for different subscripted representations for an interim or target name.

Type9: This type is a hierarchical relationship that specify adjacency of two data names in a media.

As an example, for the follwing set of MODEL statements:

```
N IS INTERIM
A0: N = 10
A1: A.B = C
A IS GROUP
B IS FIELD(A,CHAR(N))
C IS FIELD(A,CHAR(5))
```

the precedence matrix will contain the following entries:

	A	B	C	N	A0	A1
A	0	0	0	0	0	0
B	2	0	0	0	0	0
C	2	0	0	0	0	3
N	0	5	0	0	0	0
A0	0	0	0	4	0	0
A1	0	4	0	0	0	0

where it is assumed that A and B are target variables and C is a source variable.

6.6 The Directed Graph.

A directed graph is a graphical representation of the precedence matrix. Each node of the directed graph represents either a data name or an assertion name. Thus, the number of nodes in the directed graph (for a model specification) is equal to the number of dictionary entries. An arc is drawn between the two nodes that represent I th and J th dictionary entries, if the J th dictionary entry is a successor of the I th dictionary entry. The label on each arc represents the type of relationship between the two nodes.

As an example, the directed graph for the simple example given in the previous section is shown in Figure 6.5. In this graph, circles represent variable nodes, and the squares represent assertion nodes.

In general, a node of a directed graph can represent an array of elements which will be represented by the following notation:

A(l,u,c)

Where A is a varaible name, and l, u and c define the range of elements represented by the node. The above notation represents the following set of elements of variable name:

A(l), A(l+c), A(l+2.c), ..., A(l+k.c)

where k = (u-l)/c. Because a node can represent an array of elements, we will also call our directed graph as an <u>array graph</u>. A Formal definition of array graph is given in chapter 9.

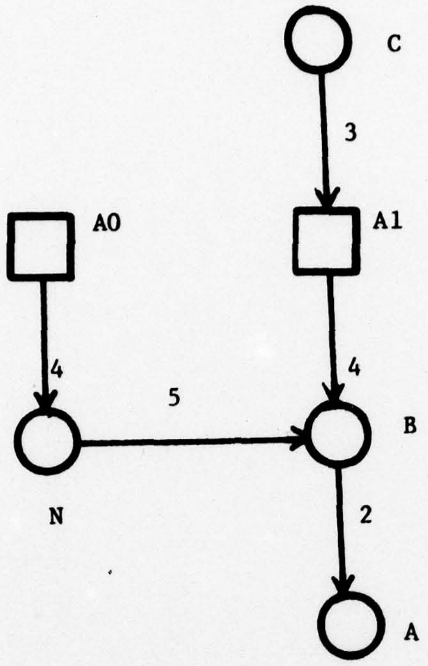


Figure 6.5. Directed Graph for the MODEL statements:

```
N IS INTERIM;
A0: N = 10;
A1: A.B = C;
A IS GROUP;
B IS FIELD(A,CHAR(N));
C IS FIELD(A,CHAR(5));
```

The directory consists of an entry for each member in the library, which consists of:

(1) name of the member,

(2) the line number of the first line of data for this member, and,

(3) the line number of the last line of data for this member.

The data area consists of input statements of all the members in the library.

When a member is added to the library, the statements for that member are padded to the end of data area; and an entry for the memebr name is added to the directory. The summary record is also updated.

When a member is deleted from the library, the statements of that member are not deleted, but the directory entry for that member is changed to a dummy entry. Also, the summary record is updated by incrementing the number of members deleted.

The library maintenance subsystem consists of two subroutines: (1) LIBCR which is used to create a new MODEL library, and (2) LIBU, which is used to update the MODEL library. In the following, these routines are described in more detail.

7.2 The LIBCR Procedure.

The LIBCR(#DIR) is the procedure to create a MODEL library. The parameter, #DIR, specifies the number of directory blocks to be used in the library. The functions of this procedure are:

(1) to write the summary record,

and (2) to write dummy directory blocks.

If no parameter is specified, then a default of two blocks is used for the directory. The number of directory entries in a block can be obtained using the following formula:

# of directory entries per block =

(block size -2)/27.

Block size is usually the track length of the medium in which the library resides. As an example, if block size is 2048, then the above formula yields 75 directory entries in a directory block. The library is organized as a Regional(1) data set.

7.3 The LIBU Procedure.

The LIBU procedure is used to update the MODEL library. It can perform the following functions:

(1). Add a member to the library.

(2). Delete a member from the library.

(3). Dump the contents of the library.

and (4). Compress the library and create a new one after eliminating all the holes (created by the deletion of members) in the library.

Figure 7.1 shows a functional overview of the LIBU procedure. It obtains input primarily from SYSIN file. The sysin file consists of two types of statements: (1) data statements and (2) control statements. A statement in the SYSIN file is a control statement if it is one of the following types:

(1). ␢ADD member-name

(2). ␢DELETE member-name

(3). ␢DUMP

(4). ␢COMPRESS

(5). ␢SEC section-name

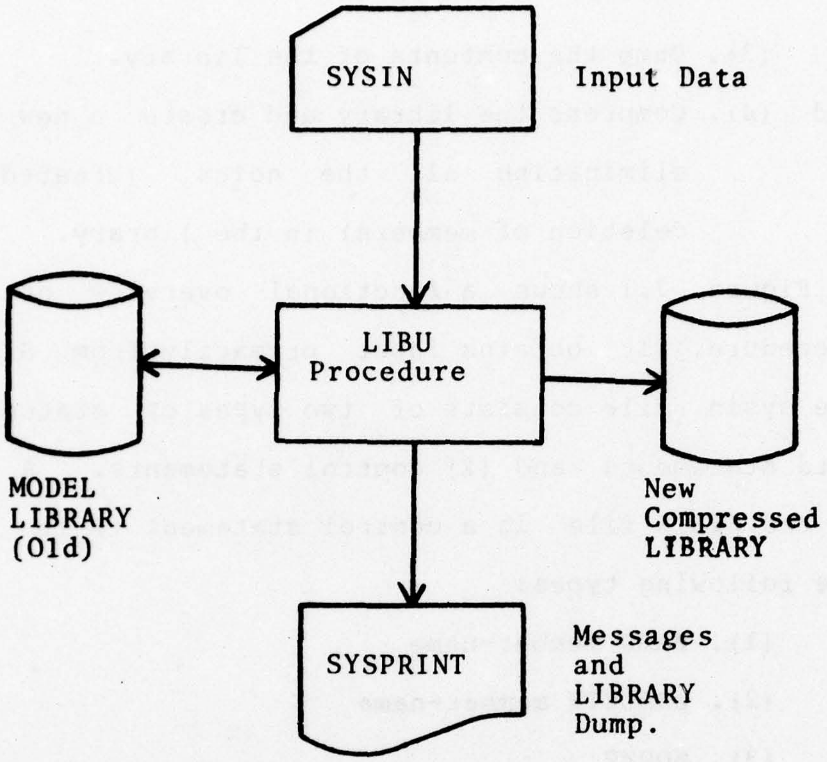


Figure 7.1. Overview of LIBU Procedure.

The first four types of control statements perform the four functions mentioned in the beginning of the section. The last control statement is used to divide the input statements into sections. This division allows the REFER statements of MODEL to include a subset of statements of the member (a section). All other statements are assumed to be the input statements. Data statements are associated with only the ADD control statement. Therefore, if an ADD statement is followed by a set of data statements (and SEC control statements), then those set of statements are saved in the library with the member name specified in the ADD statement.

When the SYSIN file contains a COMPRESS statement, each member in the MODEL library is copied to a new library as shown in Figure 7.1. The new library must have been previously created by the LIBCR procedure described in section 7.2. The reason for creating a new library rather than updating the old one itself is to guard against any system failure during the compress operation, which could clutter the library. Therfore, to compress a library, the following steps must be followed:

(1). Create a new library using the procedure LIBCR.

(2). Compress the old library and update the new library using the procedure LIBU.

(3). If the compress operation was successful in step (2), scratch the old library, and rename the new library name appropriately.

(4). If compress was not successful for some reason, repeat steps (1) thru (4) again.

```
 ADD ASS1
ASS1: A = B + C;
 ADD INT
INT IS GROUP;
  A IS FIELD(INT);
  B IS FIELD(INT);
 DELETE OLDMEM
 ADD NEWA
NEWA: A = X * Y;
```

Figure 7.2. A Sample Input for LIBU Procedure.

Figure 7.2 shows a sample input for the LIBU procedure. In this example, the procedure LIBU will add 3 members, ASS1, INT, and NEWA to the library and deletes the member OLDMEM from the library. Note that all the control statements start at column 2.

The routines LIBCR and LIBU are not part of MODEL system. They are separate stand alone procedures. However, the MODEL library can be accessed by the MODEL system during the syntax analysis phase (while processing REFER statements) and updated during the documentation phase, when the complete MODEL specification is saved as a member in the library.

CHAPTER 8

DETERMINATION OF PRECEDENCE RELATIONSHIPS

Once the syntax analysis is complete, the statements stored in the associative memory are analyzed in Phase-2 of the MODEL Processor. This analysis produces (1) a Dictionary of all names used in the MODEL specification and (2) a Precedence Matrix that represents the precedence relationships between different dictionary entries.

The dictionary is similar to the directory of the associative memory, but contains more information than the directory. There can be more than one dictionary entry for a data name represented by a directory entry. There exists one dictionary entry for each node in the directed graph that represents the relationships between different data items used in the specification.

The organization of dictionary and the Precedence Matrix was explained in chapter 6. Figure 8.1 gives an overview of the Phase-2 of the MODEL Processor. It is shown as consisting of two parts:

(1). Generation of dictionary,

and (2). Generation of Precedence Matrix.

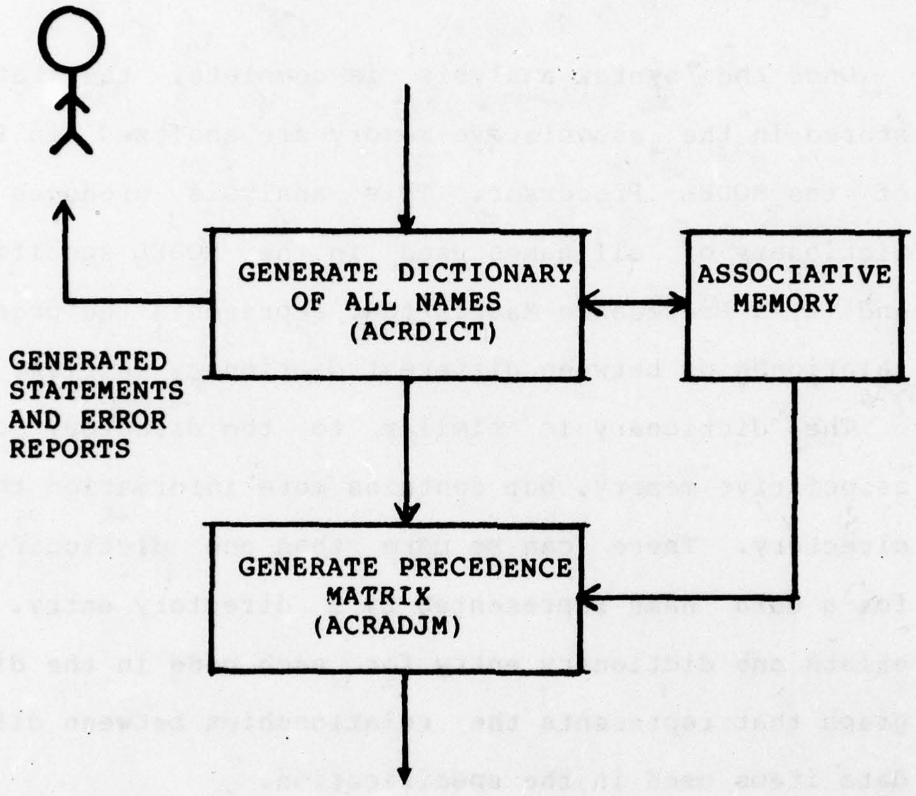

GENERATE DICTIONARY OF ALL NAMES (ACRDICT)
ASSOCIATIVE MEMORY
GENERATED STATEMENTS AND ERROR REPORTS
GENERATE PRECEDENCE MATRIX (ACRADJM)

Figure 8.1. Overview of Phase-2 of MODEL Processor.

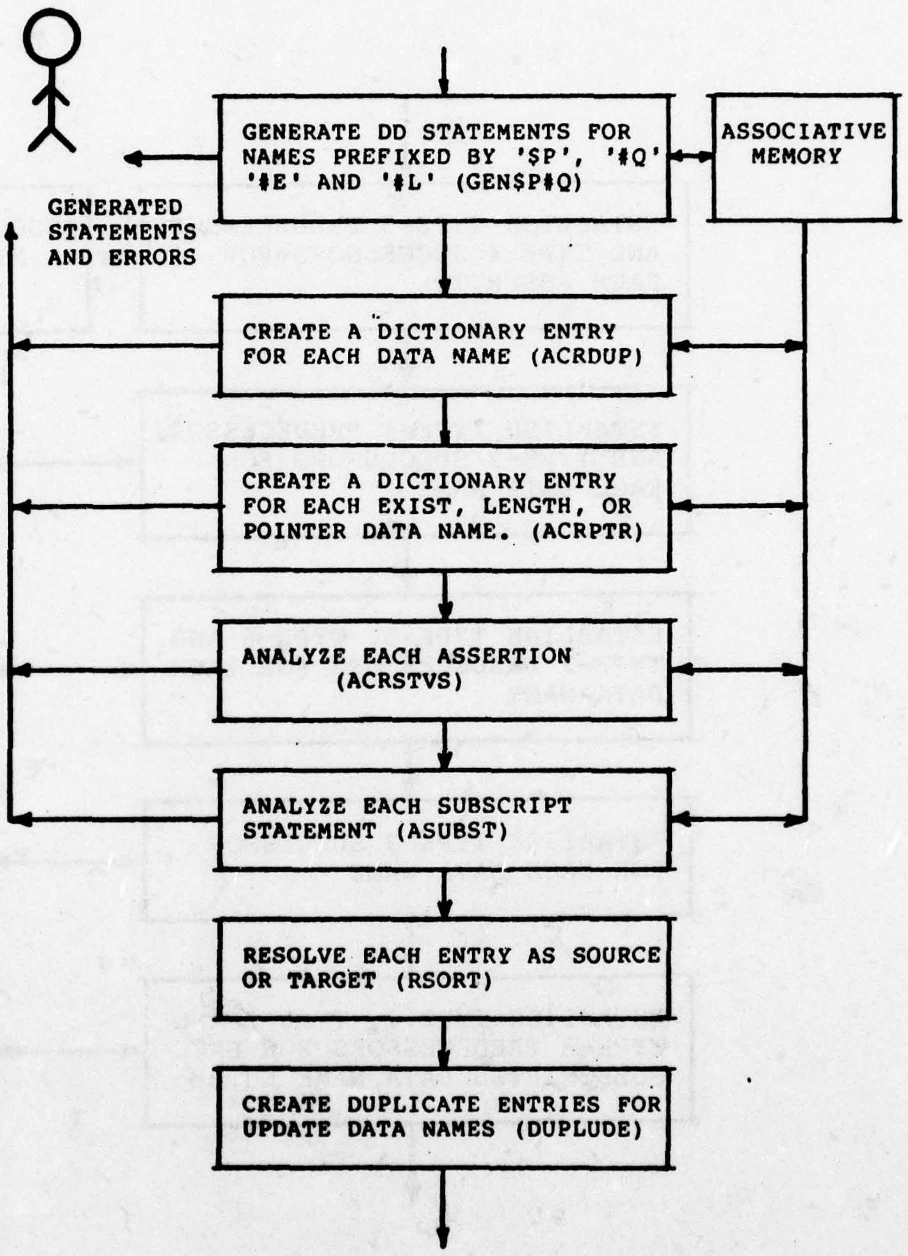

GENERATE DD STATEMENTS FOR NAMES PREFIXED BY '$P', '#Q' '#E' AND '#L' (GEN$P#Q)
ASSOCIATIVE MEMORY
GENERATED STATEMENTS AND ERRORS
CREATE A DICTIONARY ENTRY FOR EACH DATA NAME (ACRDUP)
CREATE A DICTIONARY ENTRY FOR EACH EXIST, LENGTH, OR POINTER DATA NAME. (ACRPTR)
ANALYZE EACH ASSERTION (ACRSTVS)
ANALYZE EACH SUBSCRIPT STATEMENT (ASUBST)
RESOLVE EACH ENTRY AS SOURCE OR TARGET (RSORT)
CREATE DUPLICATE ENTRIES FOR UPDATE DATA NAMES (DUPLUDE)

Figure 8.2. Overview of generating the Dictionary. (Procedure ACRDICT).

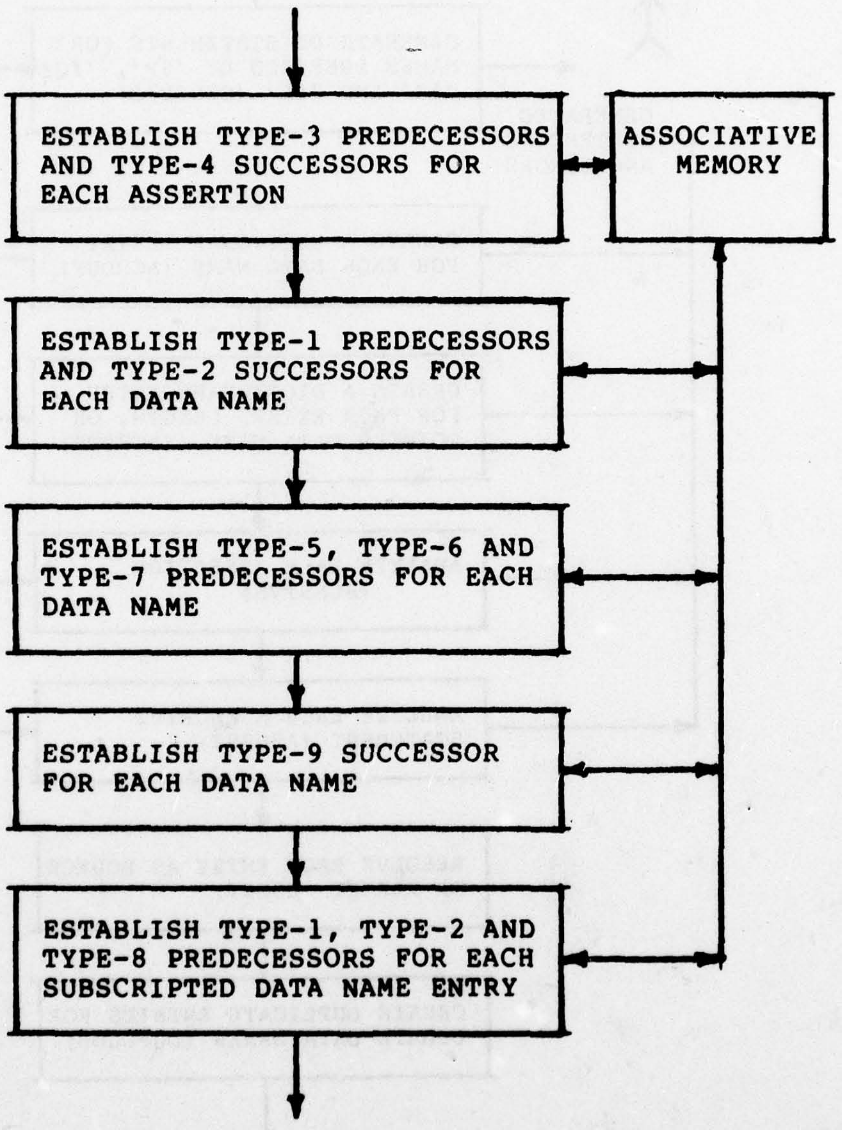

ESTABLISH TYPE-3 PREDECESSORS AND TYPE-4 SUCCESSORS FOR EACH ASSERTION
ASSOCIATIVE MEMORY
ESTABLISH TYPE-1 PREDECESSORS AND TYPE-2 SUCCESSORS FOR EACH DATA NAME
ESTABLISH TYPE-5, TYPE-6 AND TYPE-7 PREDECESSORS FOR EACH DATA NAME
ESTABLISH TYPE-9 SUCCESSOR FOR EACH DATA NAME
ESTABLISH TYPE-1, TYPE-2 AND TYPE-8 PREDECESSORS FOR EACH SUBSCRIPTED DATA NAME ENTRY

Figure 8.3. Overview of generating Precedence Matrix (Procedure ACRADJM).

Figure 8.2 shows the different steps involved in creating the dictionary. And, Figure 8.3 shows the different steps involved in creating the Precedence Matrix. The functions of different steps shown in Figures 8.2 and 8.3 are described in the following sections.

8.1 Generating the Dictionary (Procedure ACRDICT).

The first step during the network analysis is to generate a dictionary of all names supplied by the user. The names include data names and the assertion names. Algorithm A8.1 describes the routine ACRDICT which is used to create the dictionary from the MODEL specification. It uses the contents of the associative memory created during the syntax analysis phase. The process of generating the dictionary consists of the following steps as shown in Figure 8.2:

(1). Generating dd statements for the parent names for which no dd statements were specified.

(2). Generating data description statements for the successor of group and record names for which no successor exists.

(3). Creating a dictionary entry for each data name and assertion name in the directory.

(4). Creating a dictionary entry for each data name used in the assertion, after generating an appropriate dd statement, if necessary.

(5). Analyzing each subscript statement and associating a parent name for that name.

(6). Resolving each data name as source or target.

(7). Generating duplicate dictionary entries for update data names (names that are descendents of an update file).

Also, some pointers in the data area of each statement are updated. These pointers establish links between the data name, its parent, and its daughters. The dictionary creation starts first with media names, then file names, record names, group names, field names, interim names, subscript names and finally, the assertions. This type of ordering allows easy linking of data names with their parent.

During the analysis of an assertion (in step 4 of A8.1) some additional assertions could be generated (for example, while simplifying the subscripts to the "standard form"). The storage entry pointers of these generated assertions are saved in a stack. If the stack is not empty, those generated assertions are also analyzed and the source and target variables used in them are added to the dictionary (step 7 in A8.1).

The routines GEN$P#Q, ACRDUP, ACRPTR, ACRSTVS, ASUBST, RSORT and DUPLUDE used in algorithm A8.1 are described in greater detail in the sections that follow.

Algorithm: A8.1, <u>ACRDICT</u>.

Function: Creates a dictionary of all names used in the MODEL specification.

Calls: GEN$P#Q, ACRSTVS, ACRPTR, ACRDUP, ASUBST, RSORT, DUPLUDE.

Called by: MONITOR.

Step1: Using the routine GEN$P#Q, generate dd statements for all those names that are prefixed with '$P', '#Q', '#L', or '#E', if no dd statement exists for those names.

Step2: Using the routine RETREVE, get all the storage entries for the data names and the assertion names (that is, media, file, record, group, field, interim, and assertion names, in that order). For each storage entry thus obtained, perform steps 3 thru 6.

Step3: Using the routine ACRDUP, add the data name to the dictionary.

Step4: If the data name is an assertion name, then, using the routine ACRSTVS, analyze the assertion and get a list of all source and target variables used in the assertion; and also, add each of the name in the list to the dictionary.

Step5: If the data name is not an assertion name, then, using the routine ACRPTR, obtain all the length, exist and pointer names associated with the data name and add them to the dictionary.

Step6: Continue.

Step7: If any assertion is generated during the analysis of the assertion in step 4, then the storage entry pointers of those generated assertions are saved in a stack. If the stack is empty then return; otherwise, perform step 4, for each storage entry in the stack.

Step8: Now, using the routine RETREVE, get the storage entries of all subscript statements. Analyze each obtained entry using the routine ASUBST.

Step9: Using the routine RSORT, resolve each data name as source, target, or interim.

Step10: Using the routine DUPLUDE, create duplicate dictionary entries for "update" data names (data names that are both source and target).

Step11: Return.

8.1.1 <u>The GEN$P#Q Procedure</u>.

If no dd statement exists for a parent name of a data name, then the MODEL specification is incomplete. In such cases, an appropriate dd statement is generated for the parent name. For the parent of a file, a media statement is generated; in all other cases, a group statement is generated. As an example, if the following statements were specified:

```
A IS FIELD(B)
B IS GROUP(C)
```

and no dd statement for the data name C was specified, then the following group statement is generated:

```
C IS GROUP
```

Note that this statement does not have any parent, this incompleteness is resolved later (see the description of the routine RESIC).

The routine GEN$P#Q does the above function of generating data description statements for the parent names. The algorithm A8.2 describes the functions of the routine GEN$P#Q.

The routine GENSTMT, used in step 4 of the algorithm is described in the following.

Algorithm A8.2: <u>GEN$P#Q</u>.

Function: Generates dd statements for parent names of data names for which no dd statements were specified.

Calls: RETRPRX, RETRNAM, GENSTMT and RETREVE

Called by: ACRDICT.

Step1: Using the routine RETRPRX, get all the names in the directory that are prefixed by either '$P' or '#Q'. Save all such names in NAM_LIST.

Step2: Perform steps 3 thru 5 for I = 1 to # of names in NAM_LIST.

Step3: Using the routine RETRNAM, check if the Ith name in NAM_LIST has a dd statement. If there is, then goto step 5.

Step4: Using the routine RETREVE, find the statement in which the NAM_LIST(I) is defined. If the statement is a file statement, then generate a Media statement, otherwise, if the name is prefixed by '#L', or '#E', then generate an interim statement for the name, otherwise, generate a group statement for the name NAM_LIST(I). Use the routine GENSTMT for generating the dd statement.

Step5: Continue.

Step6: Return.

8.1.1.1 <u>The GENSTMT Procedure</u> (<u>Generate DD Statement</u>).

The GENSTMT(N,T,ST,P) procedure is used to generate a dd statement for the name N, the parent name of which is given by P. The type of DD statement to be generated is given by T. ST is 'S' or 'T' according as the name N is a source name or a target name respectively. The functions of the routine GENSTMT are summarized in the following:

(1). If T is 'C', 'G', 'I', 'L' or 'B' then the statement to be generated is for a record, group, interim, field or subscript name respectively. All these types of statements have the same data area (DFIELD, see the description of the extended data area in section 6.1.4.5). Therefore, DFIELD is allocated and the entries in it are set to the appropriate default values.

(2). If T is 'F' or 'M', then the statement to be generated is a file or media statement respectively. These two types of statements have the same data area (DMEDIA, see the section 6.1.4.4). Therefore, DMEDIA is allocated and the entries in it are set to the appropriate default values.

(3). Using the routine STORE, the statement is stored in the associative memory, using P as the parent name.

8.1.2 The ACRDUP Procedure (Update Dictionary).

The function of the routine ACRDUP is to update the dictionary and also update some pointers in the data area of the statement. The algorithm A8.3 describes the functions of the routine ACRDUP. They include the following:

(1) Adding the data name itself to the dictionary,

(2) Establishing link between the data name and its parent, and

(3) Determining the data name as either source or target, if that name is a descendant of a source or a target file respectively.

As an example, if the storage pointer represents the following statement:

A IS FIELD(B)

then the routine ACRDUP does the following functions.

(1) It creates a dictionary entry for A (step 1).

(2) It obtains the parent of the data name A (the name B) in step 2 and the storage entry pointer to that parent name in step 3. Note that there must be at least one dd statement for B (If there weren't any, a group statement

for B must have been generated while executing the routine GEN$P#Q; see the description of algorithm A8.2).

(3) If there was only one dd statement for B, storage entry pointer of B is stored in storage-ptr -> data-pt -> data-list -> parent-ptr (see foot note(1) and also the description of common data area in section 6.1.4.1). Also, the list of daughter names given by the dtr-ptr of the parent name (storage-ptr of B -> dtr-ptr) is updated (in the routine SETDTRP) to include the newly obtained daughter name given by storage-ptr.

(4) If, for example, the data name B is a descendant of a source file, then A is also a descendant of that source file, then the bit storage-ptr -> SRC_BY_DDS is set (in step 10) to indicate that this data name is a source data name.

The term "closest" used in step 7 is defined as follows:

If there are n parent names for the data name and u(i) (i = 1 to n) is the user-line# of the i th parent name, then the "closest" parent of the data name is either:

---

(1) A -> B -> C will be used from now on, to refer to the variable C, based on B which in turn is based on A. Note that this is an extended PL/I notation, and if there is no confusion, the above notation will be abbreviated to A -> C; in other words, the expression storage-ptr -> data-pt -> data-list -> parent-ptr will be abbreviated to storage-ptr -> parent-ptr.

(1) the i th parent name such that

$u(i) < u$ and

$\forall j = 1$ to n, $j \neq i$, $u(j) < u(i)$ or $u(j) > u$

or

(2) the i th parent name such that

$u < u(i)$ and

$\forall j = 1$ to n, $j \neq i$, $u(j) > u(i)$.

where u is the user-line# of the data name given by storage-ptr. In the above example, if the user-line# of A is 10, and B has two dd statements with the user-line#s 5 and 15 respectively, then the first parent name is selected as the parent of A (in step 7). This type of situation can happen as shown in the following set of statements:

STATEMENT	USER-LINE#
B IS GROUP(R1)	5
A IS FIELD(B)	10
B IS GROUP(R2)	15
A IS FIELD(B)	20

In this case, we can safely assume that the second and the fourth statements really mean:

A IS FIELD(R1.B)

and   A IS FIELD(R2.B)
respectively. This type of resolving the ambiguity is precisely the function of step 7 in the algorithm A8.3.

The routines GETQNM and SETDTRP used in the algorithm is described in the following sections.

The ACRCP(QNAME,#I) procedure is a local procedure that eliminates all the storage entry pointers in DIRTEMP that do not contain the name, QNAME. The second argument is decremented by the number of storage entries that do not contain QNAME. For example, if QNAME is A.B and a storage entry in DIRTEMP represents the name C.A.B.X, then QNAME is contained in it, whereas, if the storage entry represents the name C.B.A.X, then QNAME is not contained in it, so, that storage entry is deleted from DIRTEMP and #I is reduced by 1.

Algorithm: A8.3, <u>ACRDUP</u>(storage-ptr)

Function: Adds the names in the storage entry (given by storage-ptr) to the dictionary.

Calls: ACRDO, GETQNM, RETRNAM, RETREVE, ACRCP and SETDTRP.

Called by: ACRDICT.

Step1: Using the routine ACRDO, add the data name represented by storage-ptr to the dictionary.

Step2: Using the routine GETQNM, get the parent names, if any, in the storage entry. If no parent name, then goto step 9.

Step3: Using the routine RETRNAM and RETREVE, get all the storage entry pointers for the parent name. Save those pointers in DIRTEMP.

Step4: Using the routine ACRCP, eliminate all those pointers in DIRTEMP, for which the parent name represented by that pointer does not match with the parent name obtained in step 2.

Step5: If there is no pointer in DIRTEMP then goto step 9. If more than one pointer exists in DIRTEMP, then goto step 7.

Step6: DIRTEMP(1) gives the storage pointer to the data name. Using the routine SETDTRP, establish the links between parent and daughter names. At end, goto step 9.

Step7: The data description for this name is ambiguous, that is, there are more than one dd statements for its parent. In this case, select that parent which is "closest" to it.

Step8: Using the routine SETDTRP, establish the links between the parent and daughter names.

Step9: If the storage entry represents a file name then check if it is in a source file or target file statement. Accordingly, set the SRC_BY_DDS or TAR_BY_DDS bit in the storage entry of the data name.

Step10: If the storage entry represnets a group, field, or record name then check if it is a descendant of a source or a target file. Accordingly, set the corresponding bit in the storage entry of the data name.

Step11: Return.

8.1.2.1 The GETQNM Procedure (Get Qualified Name).

The function GETQNM(PTR,X) returns the fully qualified name in the storage entry (given by the storage entry pointer, PTR), the prefix of which is the same as the second argument, X. For example, if the storage entry contains the key names '$G', '$GX', '#EA', '$PY' AND '#QZ', in that order (this storage entry represents the statement: X IS GROUP(Y.Z,(A))), then the following use of the function:

```
XY = GETQNM(PTR,'$P')
```

will return the qualified name '$PY #QZ' in XY; and the following use of the function:

```
XY = GETQNM(PTR,'#E')
```

will return the name '#EA' in XY.

8.1.2.2 <u>The SETDTRP Procedure (Set Dtr-Parent Link)</u>.

The SETDTRP(P,D) procedure is used to establish the link between the storage entry pointer P (parent of D) and the storage entry pointer D (daughter of P). As described earlier, the storage entry pointers of all daughter names of a data name are maintained in a list given by dtr-ptr of the storage entry (see the description of common data area in section 4.5). The routine SETDTRP adds the storage entry pointer D to the daughter list of P such that the user-line#s of all the entries in the list are in ascending order. For example, if the list contained 2 data names with user-line#s 5 and 10 respectively, and the user-line# of D is 8, then the storage entry D is added in between the two data names; however, if the user-line# of D is 20, then the storage entry D is added to the end of the list.

8.1.2.3 <u>The ACRPTR Procedure (Analyze PTR Variables)</u>.

The ACRPTR(PTR) procedure adds the exist names, length names and the key names (if any) associated with the storage entry (given by the storage entry pointer, PTR) to the dictionary. The algorithm A8.4 describes the functions of the routine ACRPTR. For example, if PTR represents the storage entry pointer corresponding to the following statement:

```
        A IS FIELD(B,(R1,R2),CHAR(L1))
```

and the data names R1, R2 and L1 are not descendants of the same file as that of A, or the data names R1, R2 and L1 appear after the data name A (that is, user-line#s of R1, R2 and L1 are greater than the user-line# of A), then the following interim dd statements are generated:

```
        R1 IS INTERIM(EXIST.B.A)
        R2 IS INTERIM(EXIST.B.A)
        L1 IS INTERIM(LEN.B.A)
```

If however, any of the names R1, R2 or L1 occurs before the data name A and is the descendant of the same file as A, then there is no necessity of generating a dd statement for that name, because, such a definition can directly be handled using the "REFER" option of PL/I Optimizing Compiler. This is further described in the later chapters.

If there is a key name in the dd statement (storage entry pointer to which is given by PTR), an interim statement for the pointer name is always generated.

The routine SRCHFIL used in step 4, searches the length or exist name in the file structure and returns '1'B is the search is successful, '0'B, otherwise.

The routine GENQST, used in step 6 of the algorithm is described in the following sub-section.

Algorithm: A8.4, ACRPTR(PTR).

Function: Adds to the dictionary the length, exist and pointer names in the dd statement.

Calls: GENQST, SRCHFIL.

Called by: ACRDICT.

Step1: Get all the key names in the storage entry. For each key name generate the following statement using the routine GENQST:

key-name IS INTERIM (POINTER.qual-name)

where qual-name is the qualified name that the storage entry (given by PTR) represents.

Step2: For each length or exist name in the dd statement perform steps 3 thru 7.

Step3: If the name is defined by '*' then goto step 6.

Step4: Using the routine SRCHFIL, see if the name is in the same file (as the name given by PTR), and that name (exist or length) occurs before it. If it is, then goto step 7.

Step6: Generate the following DD statement:

name IS INTERIM (prefix.qual-name)

where "name" is the exist or the length name, and prefix is 'EXIST' or 'LEN' according as "name" is an exist name or length name respectively.

"qual-name" is the qualified name that the storage entry (given by PTR) represents. Use the routine GENQST to generate the statement.

Step7: Continue.

Step8: Return.

8.1.2.4 <u>The GENQST Procedure</u>.

The GENQST(N,I,P) procedure is used to generate a dd statement for all the names in the qualified name N. I specifies the type of data name that N represents. It is one of the following names: 'EXIST', 'LENGTH', 'POINTER', 'SOURCE', 'TARGET', 'NEW', 'OLD' or 'SUBSET'.

The storage entry pointer of the last name in N is returned in the third argument, P. Algorithm A8.5 describes the functions of the procedure GENQST.

Note that, because the type of data name is known, the parent of the data name is also known (which is one of the group names: 'EXIST', 'LENGTH', ... 'SUBSET'). Thus, in the algorithm A8.5, it is easy to search for a dd statement of a simple name in N, by searching the daughter list of its parent name. For example, if I = 'EXIST' and N is 'REP1' then the daughter list of the data name '$GEXIST' is searched for the name REP1.

Algorithm: A8.5, GENQST(N,I,P).

Function: Generates dd statements for the names in the qualified name in N, the type of which is given by the second argument.

Calls: ACRDO, GENSTMT and SETDTRP.

Called by: ACRPTR.

Step1: Generate a group statement for the Ith data name, if this data name is the first one in that type. Use the routine to generate the dd statement and the routine ACRDO to add the name to the dictionary.

Step2: For each simple name in N perform steps 3 thru 6.

Step3: Check if there exists a dd statement for the simple name (A dd statement exists if the simple name is contained in the daughter list of its parent). If there is a dd statement already, then goto step 6.

Step4: Generate a dd statement for the simple name using the routine GENSTMT. Also, add the name to the dictionary using the routine ACRDO.

Step5: Establish the link between daughter and parent names using the routine SETDTRP.

Step6: Continue.

Step7: Return.

8.1.3 <u>The</u> <u>ACRSTVS</u> <u>Procedure</u> (<u>Analyze</u> <u>Vars.</u> <u>in</u> <u>Assertion</u>).

The ACRSTVS(PTR) procedure analyzes the assertion, the storage entry pointer of which is given by the argument, PTR. It performs the following functions:

(1). Obtains all the source and target variables in the assertion.

(2). If necessary, generates dd statements for the data names used in the assertion.

(3). Analyzes the subscript of each data name, and, if necessary, simplifies the assertion by reducing the subscript of each data name to the standard form: "a*I+b", where "a" and "b" are constants (or simple variable names) and I is a subscript name. This process of simplifying assertions can also cause the generation of some additional assertions. (See also the description of the routine GET_AE).

(4). Obtains the lower bound, upper bound, and increment for each subscript of a data name. These are obtained from the subscript itself and the range specified in the "if-clause" of the assertion.

The algorithm A8.6 describes the functions of the routine ACRSTVS in greater detail.

Each variable that was found in the assertion is stored in a list, the format of which is described in section 6.2.1.

The routine GET_AE in step1 of algorithm A8.6, sets the lower bound (lb) and upper bound (ub) of a subscript to the offset (off) specified. These values are changed in step 6A, where the entries lb, ub and inc are computed using the routine MERGSUB, which uses the information specified in the dd statement for the data name and the "if-clause" of the assertion. The routine MERGSUB is described in section 8.1.3.3.

Algorithm: A8.6. <u>ACRSTVS</u>(S).

Function: Analyzes the assertion, the storage entry pointer of which is given by the argument, S.

Calls: GET_AE, GENSTMT, SETDTRP, ACRD1, MERGSUB.

Called by: ACRDICT.

Step1: Using the routine GET_AE, get all the source and target variables in the assertion. A list of all the variables are then created, and the pointer to that list is saved in the pointer "vl-ptr", located in the extended data area of the assertion (see also the section 6.1.4.3).

Step2: For each variable obtained in step1, perform steps

3 thru 10.

Step3: For each simple name in the variable name, perform steps 4 thru 9.

Step4: If the simple name does not have a dd statement, then generate a dd statement for that name using the routine GENSTMT. If the variable name contains only one simple name, then an interim name is generated, otherwise, a group statement is generated except for the last simple name, for which a field statement is generated.

Step5: If there is more than one dd statements for the simple name, then select the one with "matching" parent name.

Step6: Using the routine SETDTRP, establish links between the simple name and its parent.

Step6a: Using the routine MERGSUB, obtain the lower bound (lb), upper bound (ub) and increment (inc) for the subscripted variable.

Step7: Using the routine ACRD1, create an entry for the simple name in the dictionary.

Step8: Set the bit SRC_BY_ASS or the bit TAR_BY_ASS according as the variable name is the source or target of the assertion, respectively.

Step9: Continue.

Step10: Continue.

Step11: Return.

8.1.3.1 <u>The GET-AE Procedure</u> (<u>Get Vars</u>. <u>in Arith Expn.</u>).

The GET_AE(S) procedure obtains all the variables in the arithmetic expression of an assertion. The argument, S, points to the DT for the arithmetic expression. The main functions of the procedure GET_AE are outlined in the algorithm A8.7. In the following, the algorithm is illustrated with some examples.

One of the main functions of the routine GET_AE is to simplify the assertion such that any subscript associated with a variable name is in the standard form: "a*I+b" (Step 4). For example, in the following assertion:

```
AB(20*A+B+D) = C(I)
```

the subscript of AB is not in the standard form. This subscript can be reduced to the standard form, once the data types of the variable names A, B and D are known. In the following, three cases are considered.

Case(1) If none of the variables names A, B and D is defined as a subscript, then the subscript is reduced to the standard form by generating the following assertion:

```
$I0001 = 20*A+B+D
```

and changing the original assertion to:

AB($I0001) = C(I)

(in this case, a = 0 and b = $I0001, in the standard form of the subscript expression).

Case(2) If A was declared as a subscript, then the following assertion is generated:

$I0001 = B + D

and the original assertion is changed to:

AB(20*A+$I0001) = C(I)

Case(3) If A and D were not declared as subscripts, but B was, then the following assertion is generated:

$I0001 = 20*A+D

and the original assertion is changed to:

AB(B+$I0001) = C(I)

The functions described above are performed in step 4 of the algorithm. The general rules for simplifying a subscript are given below:

(1). Search the derivation tree for a subscript name. Any name that appears in the leaf of the tree is a subscript name if the following conditions are satisfied:

(a). The name is defined as a subscript (by a subscript statement) or the name does not have any dd statement and the first character in the name is one of the letters, 'I', 'J', 'K', 'L', 'M', or 'N'.

(b). The name does not have a subscript.

Note that, if the assertion given above is slightly changed to the following:

```
AB(20*J+B+D) = C(I)
```

the case(1) above, will generate the following assertion:

```
$I0001 = B + D
```

and the original assertion is changed to:

```
AB(20*J+$I0001) = C(I)
```

(2). If a subscript name was found, then the tree is transformed such that the subscript name is the left most "handle" in the tree. For example, if in the subscript, "20*A+B+D", B is found to be the subscript name, then the tree is changed such that it now represents the expression: "B+20*A+D".

(3). If no subscript name is found, and the subscript is not a constant (or a simple name), then an assertion is generated as in case(1) above.

(4). Now the offset and increment associated with the subscript name is determined. If the offset ("a") and increment ("b") are not constants (or simple variable names) then appropriate assertions are generated as in the cases (2) and (3) above.

Algorithm: A8.7, GET_AE(S).

Function: Obtians all the variables in the arithmetic expression represented by the argument pointer, S.

Calls: GENASSI.

Called by: ACRSTVS.

Step1: Obtain all the variable names in the arithmetic expression by scanning the DT recursively. For each variable name obtained, perform steps 2 thru 9.

Step2: For each subscript of the variable name perform steps 3 thru 8.

Step3: Check if the subscript is in the standard form: a*I+b, where "a" and "b" are constants (or simple variable names) and "I" is a subscript name. If the subscript is in the standard form then goto step 7.

Step4: Reduce the subscript to the standard form by generating additional assertions. Use the routine GENASSI to generate assertions. This reduction procedure can generate at most two assertions of the following form:

a = <arith-expression>

and b = <arith-expression>

Step5: If the subscript is a constant, then set the corresponding bit (bits(6), see the description of VAR-LIST in section 8.1.4) in VAR-LIST.

Step6: If the subscript is "misc-data" (float data or string) then set the corresponding bit (bits(5)) in VAR-LIST.

Step7: Get the offset and increment for the subscript and save it in OFF and INC (of VAR-LIST) respectively.

Step8: Continue.

Step9: Continue.

Step10: Return.

8.1.3.2 <u>The GENASSI Procedure</u> (<u>Generate Assertions</u>).

The GENASSI(P,L) procedure is used to generate an assertion during the simplification of a subscript. The first argument, P, points to the DT of an arithmetic expression, or a diadic expression depending on the second parameter L. Algorithm 8.8 describes the functions of the routine GENASSI.

Algorithm: 8.8, GENASSI(P,L).

Function: To generate an assertion for the expression represented by P and L. P points to an arithmetic expression or a diadic expression depending on the second argument L.

Calls: STORE.

Called by: GET_AE.

Step1: Get an interim name (name prefixed by '$I' and followed by a 4 digit number, example, $I0001) and an assertion name ('$A' concatenated with the interim name).

Step2: Generate dd statement for the interim statement obtained in step 1.

Step3: Allocate entries for "assertion", "simple assertion", and "arithmetic expression" and link them appropriately, such that the expression given by the argument P is the RHS of the assertion and the interim name obtained in step 1 as the LHS of the assertion. Also, change P to an arithmetic expression that consists of just the interim name obtained in step 1.

Step4: Using the routine STORE, store the assertion in the associative memory. Also, stack the storage entry pointer for the assertion, so that it can be analyzed using the routine ACRSTVS (see the description of routine ACRDICT).

Step5: Return.

8.1.3.3 <u>The MERGSUB Procedure (Merge Subscript Ranges)</u>.

The MERGSUB(P) procedure obtains the subscript ranges (lower bound, upper bound and increment) for all dimensions of a variable. VARLISTP points to the structure VAR_LIST which contains the information about a varaible used in an assertion (the root of the DT of which is given by P). The algorithm A8.9 describes the functions of the routine MERGSUB. In the following, the routine MERGSUB is illustrated with some examples.

In Step1 of the algorithm, the dimension of the variable under considersion is obtained from the dd statement of the variable name and that of its parent. For example, if the following dd statements were given:

```
A IS GROUP(X,(5))
    B IS FIELD(A,(10))
```

Then B is recognized as a variable of dimension 2 and size 5x10.

If the dimension computed in step1 is greater than the number of subscripts specified for the variable (#OFI in the structure VAR_LIST), then some of the subscripts in the assertion have been ommitted. In this case, the structure VAR_LIST is updated to contain the maximum number of subscripts (determined in step 1). For example, if VAR_LIST represents the subscripted variable B(3), and B was described as of having 2 dimensions (with size = 5x10, as above), then the structure VAR_LIST is changed to represent the subscripted variable B(*,3).

If no range for a subscript was specified in the "if-clause" of the assertion (that is, there does not exist any if-clause for the assertion, or the if-clause contains just a boolean expression, and not a subscript expression), then the lower bound and upper bound for each dimension of the variable is obtained from the dimension and size of the variable obtained in step 1.

If a range for a subscript is specified in the if-clause of the assertion, then that range is added to the lower and upper bounds in lb and ub. For example, if a variable was subscripted by "2*I+5", then the lower and upper bounds for the corresponding dimension would be both 7. If the if-clause of the assertion in which the above subscripted variable occurs is of the form: "For I = 1 to 5", then the lower and upper bounds (lb and ub) for the corresponding variable are changed to 7 and 15 respectively. These computaions are done in step 7 of the algorithm. The increment for the subscript is also computed in step 7 by multiplying the value specified in the subscript with the value given in the if-clause. In the above example, the increment is 2 (the increment specified in the if-clause is 1). However, if the if-clause in the assertion is changed to "For I = 1 to 5 BY 2" then the increment is 4 (note that, lb and ub in this case are 7 and 13 respectively).

Algorithm: A8.9, MERGSUB(ASSPTR,STEPTR)

Function: Obtains the lower and upper bounds and the increment associated with any subscript of a variable used in an assertion. ASSPTR points to the derivation tree of the assertion and STEPTR is the storage entry pointer of the variable name for which the bounds and the increment are to be supplied.

Calls:

Called by: ACRSTVS

Step1: Get the Dimension, #D, of the data name given by the storage entry pointer STEPTR. Also, get the bounds of each dimension in the array SIZE.

Step2: If #D < #OFI (dimension of the subscripted variable used in the asserton) then goto step 11. If #D = #OFI then goto step 5. Note that the external pointer VARLISTP points to the structure VAR_LIST that describes the variable name used in the assertion.

Step3: Create a new VAR_LIST (save the pointer in TPTR) such that it can accommodate #D dimensions. Also, move the subscript information associated with the variable to the last #OFI dimensions of the newly created VAR_LIST.

Step4: Set the default values for the first #D-#OFI subscripts (lb = 0, ub = 0 and subscript-ptr = null).

Step5: For I = 1 to #D perform steps 6 thru 9.

Step6: If the subscript name was not specified ('*' was used), or no range for that subscript was specified in the if-clause, then set the lower and upper bounds (lb and ub) for the I th dimension with the corresponding values from SIZE, obtained in step 1.

Step7: If there was a subscript range for the I th dimension, then add the lower and upper bounds specified in the if-clause to the corresponding values in VAR_LIST. Also, obtain a new increment for the subscript by multiplying the increments specified in the if-clause and in the subscript itself.

Step9: Continue.

Step10: Go to step 12.

Step11: Increase the dimension of the data name given by STEPTR by #OFI-#D and goto step 5.

Step12: Return.

8.1.4 Analysis of Subscript Statements (Procedure ASUBST).

The procedure ASUBST is used to analyze each subscript statement specified by the user and check its consistency. The function of the procedure is to check the subscript statements to satisfy the following rules:

(1). Only one statement can be specified for each subscript statement. If more than one is specified, then all those statements, except one (arbitrarily chosen), are deleted.

(2). Only one subscript name is associated with any dimension of a data name. If more than one is specified, then all those statements, except one (arbitrarily chosen), is deleted. Note that if a data name is of dimension > 1, then one subscript name can be associated with each dimension of that data name.

(3). If any parent name is specified for a subscript name, then that parent name must be a repeating data name. If not, the parent structure of the data name is searched for a repeating data name. If a repeating data name is found, then that data name is used as the parent for the subcript name; otherwise, if the data name is not an interim data name, then the inconsistency is reported as an error to the user. However, if the data name is an interim data name, its dimension is increased accordingly. As an example, consider the following set of statements:

```
G IS GROUP(10);
  A IS GROUP(G);
    B IS FIELD(A);
I IS SUBSCRIPT(B);
```

In the above example, B is parent of I, but B is not a repeating data name. However, its grand parent G is repeating, therefore the procedure ASUBST changes the subscript statement to:

```
I IS SUBSCRIPT(G);
```

8.1.5 The RSORT Procedure (Resolve as Source or Target).

The procedure RSORT is used to resolve a data name to be source, target, interim or update. This resolved information is stored in "STATUS-BITS" of the structure "data-list" described in section 6.1.4.1. The routine RSORT uses the following rules for resolving the type of data:

(1). A data name is resolved as an update data name if it is defined as an update data name (by including the parent file name in both the source file and target file statement). In this case, 8 th bit of STATUS-BITS is set.

(2). A data name is resolved as target, if either, the data name is defined as target, or if the data name is never used as source in any assertion. In this case, 6 th bit of STATUS-BITS is set.

(3). A data name is resolved as source, if either, the data name is defined as source, or if the data name is never used as target of any assertion. In this case, 5 th bit of STATUS-BITS is set.

(4). A data name is resolved as interim, if it is used as source in one assertion and as target in another. In this case, 7 th bit of STATUS-BITS is set.

When a data name is resolved as source, target, interim, or update, all its predecessors, and the successors of those predecessors are assumed to be of the same type. However, if the resolved data type conflicts with the previously resolved type, then type that has higher priority is used. The priority of different types are given in the following:

UPDATE	(highest)
INTERIM	
TARGET	
SOURCE	(lowest)

For example, if A and B are descendants of a group G, and if A is resolved as target, while, B is resolved as interim, then both are resolved to be interim data names.

8.1.6 <u>The DUPLUDE Procedure</u> (<u>Dupl.</u> <u>Update</u> <u>Dict</u> <u>Entries</u>).

This routine creates duplicate storage entries for update data names, that is, for those data names that are used both as source and target. Note that there does not exist two storage entries for each update data name (only duplicate dictionary entries). The duplicate dictionary entries are stored in INDEX1 and INDEX2 of the "DATA-LIST" structure described in section 6.1.4.1. Also, not every update data name entry is duplicated; only those entries that are updated by some assertion are duplicated. For example, consider the following set of statements:

```
SOURCE FILE: MASTER;
TARGET FILE: MASTER;
MASTER IS FILE;
   REC IS RECORD(MASTER);
        CUST# IS FIELD(REC);
        BALANCE IS FIELD(REC, ... );
        ...
BALANCE = BALANCE + 100;
```

In this example, only the name BALANCE is updated. Therefore, there will be two dictionary entries for the name BALANCE and all its predecessors (in this case, REC and MASTER). The data name CUST# is not updated, and therefore, only one entry will be provided for that name. After the creation of the precedence matrix, the directed graph for the above example will be as shown in Figure 8.4.

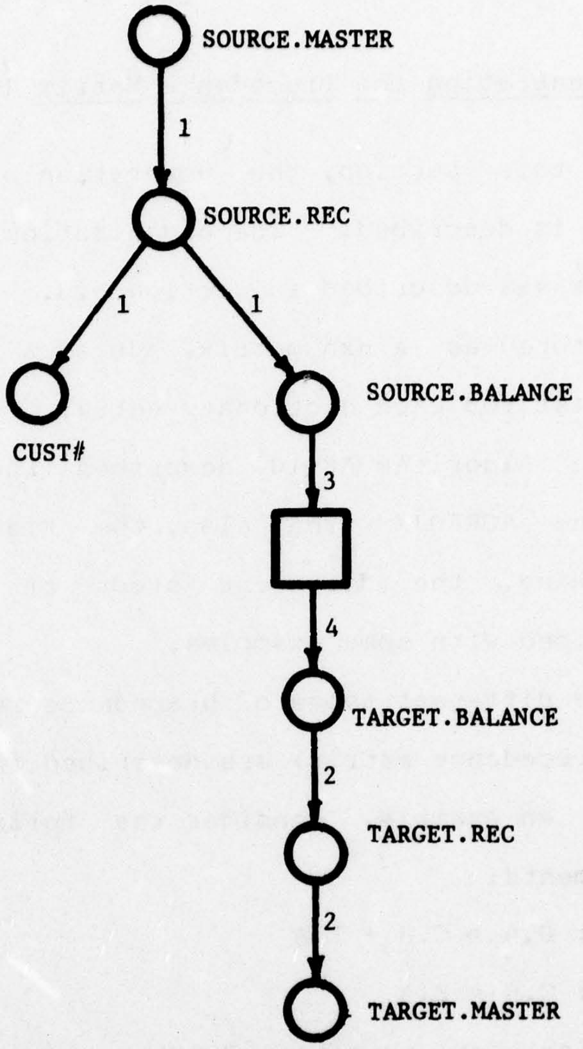


Figure 8.4. Directed Graph for a MODEL Specification Using an UPDATE File.

8.2 Generating the Precedence Matrix (Procedure ACRADJM).

In this section, the generation of precedence matrix (PM) is described. The organization of the precedence matrix was described in section 6.3. Note that the PM is not stored as a nxn matrix, but as a list of structures, one list for each dictionary entry.

The Algorithm A8.10 describes the functions of the routine ACRADJM (see also, the Figure 8.3). In the following, the different steps of the algorithm are described with some examples.

The different types of precedence relations (entries in the precedence matrix) are described in section 6.5.

As an example, consider the following set of MODEL statements:

A0: D.A = C.B + E.A

A1: C.B = E.A

Assuming that no other MODEL statements were specified, the data names C and C.B would be resolved as interim, D and D.A as target, and E and E.A as source in procedure RSORT (see section 8.4). With these assumptions, the entries (Ib,Ic), (Ia,Id) and (Ie,Ia) are set to 2, 2 and 1 respectively, where Ia, Ib, Ic, Id and Ie represent the dictionary indices of the data names A, B, C, D and E respectively. Note that, qualified interim variable names are treated the same way as the qualified target variable names. We do allow structured description of interim variables, just the same way as a source or target file structure.

All the value dependency relationships (type 3 and type 4 entries between an assertion name and the source and target variables used in that assertion) are entered in step 4 of algorithm. The hierarchical relationships (type-1 and type-2 entries between data names and their parents) are entered in step 5. The pointing relationships (types 5, 6 and 7) are entered in step 6.

To illustrate the function of step 6 in the algorithm, assume that the MODEL specification contains the following statement:

```
A IS FIELD((E),CHAR(L))
```

In this case, the dictionary contains entries for A, E and L. The function of step 6 is to set the entries (J,I) and (K,I) to 5 and 6 respectively, where I, J and K are the dictionary indices for A, E and L respectively. The function of the routine GETFNAM is to obtain the dictionary index of the storage entry that contains A and E or A and L, while processing the names E and L respectively.

If the MODEL specification contains subscripted variables, then some additional type-1 and type-2 relationships are entered in step 8 using the routine SETTYPE. This routine also enters type-8 relationships between the subscripted data names (either interim or target).

The routines GETFNAM and SETTYPE used by the routine ACRADJM are described in the sections that follow. The routines ADJMSET and ACRDR are explained in chapter 6.

8.2.1 <u>The GETFNAM Procedure</u> (<u>Get St. Entry of Name</u>).

The GETFNAM(N,I) procedure returns in I, the dictionary index of the data name that depends on the exist, length or pointer name represented by N. For example, if N is 'EXIST.A.E', then the dictionary index of A which contains the exist name 'E', is returned in I. The algorithm A8.11 describes the functions of the procedure GETFNAM. Note that if an exist, length or pointer name was specified in a dd statement of a data name, then the storage entry of that data name contains that exist, length or pointer name (prefixed by '#E', '#L' or '#K' respectively) as one of the key names.

Algorithm: A8.10, <u>ACRADJM</u>.

Function: Generates the adjacency matrix for all the entries in the dictionary.

Calls: ADJMSET, ACRDR, GETFNAM, SETTYPE, SETT56.

Called by: MONITOR.

Step1: For I = 1 to #DICT (Number of dictionary entries) perform steps 2 thru 7.

Step2: Using the routine ACRDR, obtain the storage entry pointer of the I th dictionary entry.

Step3: If the storage entry does not represent an assertion, then goto step 5.

Step4: For each source variable in the assertion set the (J,I) th entry (where J is the dictionary index of the source variable name) to 3; and for each target variable name set the (I,J) th entry (where J is the dictionary index of the target variable) to 4. Use the routine ADJMSET to set any entry of the matrix. At end, goto step 7.

Step5: If the data name has a parent, then set the (J,I) th entry or (I,J) th entry (where J is the dictionary index of the parent name) to to 1 or 2 according as the data name is a source or target name respectively. If the data type is not known, then set the (I,J) th entry to '2'.

Step6: If the data name represents length, exist, or pointer name, then set (I,J) th entry (where J is the dictionary index, obtained using the routine GETFNAM, for the data name which depends on the name represented by the I th dictionary entry) to 5, 6 or 7 respectively.

Step6a: If the data name is used in specifying length or repetition of another data name, then set type-5 or type-6 relationship between the two data names using the routine SETT56.

Step7: Continue.

Step8: Using the routine SETTYPE, set type-8 entries in the dictionary. These entries represent the relationships between different subscripted representations for a data name.

Step9: Return.

Algorithm: A8.11, GETFNAM(N,I).

Function: Obtains the dictionary index of the data name that depends on the exist, length or pointer name represented by N.

Calls: RETREVE.

Called by: ACRADJM.

Step1: Initialize I to 0.

Step2: Using the routine RETREVE, get all the storage entry pointers of the last simple name in N (prefixed by '#E', '#L' or '#K' according as the first name in N is 'EXIST', 'LENGTH' or 'POINTER' respectively).

Step3: If number of storage entries obtained in step 2 is 0 then return.

Step4: If only one storage entry was obtained in step 2, set the dictionary index of that storage entry to I and then return.

Step5: Select that storage entry that contains N (except for the first or the last name) and set I accordingly.

Step6: Return.

8.2.2 <u>The SETTYPE Procedure</u> (<u>Set Type-1, 2 & 8 Edges</u>).

The SETTYPE procedure enters the type-1, type-2 and type-8 relations between different subscripted entries of a data name. Typel relatioship is established between a source data name and all the entries that represent different subscripted use of the same data name. Type2 relationship is established between different subscripted use of a target name and the target name itself. Type8 relationship is established between different subscripted use of an interim or target data name.

For example, if the variable name A is a descendant of a source file, and an element of A is used in the following assertion:

A1: B = A(10)

then a type-1 relationship is entered in the (I,J) th entry, where I and J are the dictionary indices of A and A(10) respectively. As another example, if the variable name A is a descendent of a target file, and that variable name is used in the following assertions:

A1: A(1) = B(1)

A2: FOR I = 2 TO 10 A(I) = A(I-1)

then type-2 relationships are entered in the (J,I), (K,I) and (L,I) th entries, where I,J,K and L are the dictionary indices of A, A(1), A(I) and A(I-1) respectively. Also, in this case, type-8 relationships are entered in the (J,K) and (L,K) th entries. Type 8 relations are described in the following.

As described earlier, every subscripted variable is stored in an encoded form in the structure VAR_LIST. The encoded representation includes the lower bound, upper bound and the increment associated with the subscript. This information is useful in deriving the type-8 relationships between different subscripts of interim or target data names. If a type-8 relationship exists between two entries I and J, then it means that:

(1). I th entry is a target of some assertion (or it represents a field in a source file),

(2). J th entry is a source of some assertion (or it represents a field in a target file), and

(3). At least one element in the set of elements of the array represented by the subscript in the I th entry is contained in the set of elements represented by the subscript in the J th entry.

In the following, the representation: A(l,u,c) will be used to represent a subscripted data name A, the lower bound, upper bound and increment of which are given by l, u and c respectively. As an example, consider the following set of assertions:

```
A1: A(25) = 10
A2: FOR I = 26 TO 50 BY 1 A(I) =A(I-1)
```

The above set of assertions creates three subscripted entries in the dictionary, which are given by: A(25,25,1), A(26,50,1) and A(25,49,1) (which represent the subscripted names A(25) in assertion A1, and A(I) and A(I-1) in assertion A2 respectively). The directed graph for the above set of assertions (which includes the type-8 relations) is given in Figure 8.5.

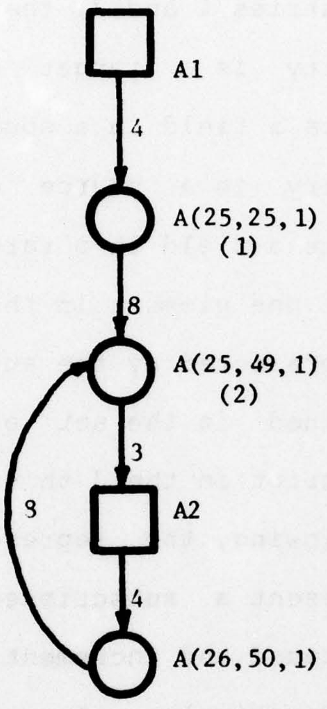


Figure 8.5. The Directed Graph for the Assertions:

A1: A(25) = 10;
A2: FOR I = 26 TO 50 A(I) = A(I-1);

The algorithm A8.12 summarizes the functions of the routine SETTYPE. The routine CHKCONT used in step 10 is described in the next section.

Algorithm: A8.12, SETTYPE.

Function: Enters type-1, type-2 and type-8 relations between subscripted data names.

Calls: CHKCONT

Called by: ACRADJM.

Step1: For I = 1 to #DICT perform steps 2 thru 13.

Step2: If the entry does not represent a name with subscript, then goto step 13.

Step3: If the I th dictionary entry represents a source data name then set the entries (I,J1), (I,J2), ... (I,Jn) to 1, where J1, J2, ... Jn are the dictionary indices of the different subscripted representations of the same name.

Step4: If the I th dictionary entry represents a target data name, then set the entries (J1,I), (J2,I), ... (Jn,I) to 2.

Step5: Let J(1), J(2), ... J(n) be the dictionary indices of n subscripted representations of the I th dictionary entry. If n < 2 then goto step 13.

Step6: For k = 1 to n perform steps 7 thru 12.

Step7: If J(k) th dictionary entry is not a target of any assertion, then goto step 12.

Step8: For 1 = 1 to k-1 & k+1 to n perform steps 9 thru 11.

Step9: If J(l) th dictionary entry is not a source of any assertion, then goto step 11.

Step10: Using the routine CHKCONT, set the adjacency matrix entry (J(k),J(l)) to 8 if at least one element represented by the entry J(k) is contained in the set of elements represented by J(l).

Step11: Continue.

Step12: Continue.

Step13: Continue.

Step14: Return.

8.2.2.1 <u>The Diophantine Equation</u>: a.x+b.y=n.

A type-8 relation exists between two nodes i and j (represented by $A(l_i, u_i, c_i)$ and $A(l_j, u_j, c_j)$ respectively, if there exists at least one set of non-negative integers (x,y) that satisfies the following relations:

$$l_i + x . c_i = l_j + y . c_j \qquad -(1)$$

$$l_i + x \cdot c_i <= u_i \qquad -(2)$$

$$l_j + y \cdot c_j <= u_j \qquad -(3)$$

Ignoring the relations (2) and (3), for the moment, the relation (1) can be rewritten as:

$$x \cdot c_i - y \cdot c_j = l_j - l_i \qquad -(4)$$

The above equation is a Diophantine equation of the form a.x+b.y=n. In the following, some of the well known results on the solution of Diophantine equation a.x+b.y=n will be sketched. Some theorems are stated without proof. The proofs can be found in standard text books on theory of numbers (GRI54).

<u>Theorem</u> 8.1.

The linear Diophantine equation a.x+b.y=n has a solution iff the greatest common divisor (GCD) of a and b divides n.

As an example, consider two nodes represented by (A,1,10,2) and A(2,10,2) respectively. The intersection of the elements represented by these nodes is empty; therefore, the Diophantine equation:

$$1 + m.\ 2 = 2 + n\ .\ 2$$

should not have any solution. The above equation can be rewritten as follows:

$$2 . m - 2 . n = 1$$

This equation does not satisfy Theorem 8.1, because GCD(2,2) = 2 which does not divide 1. Therefore, there does not exist any solution for the above Diophantine equation.

<u>Definition</u>: Two or more integers are <u>prime to other</u>, or <u>relatively prime</u> if their greatest common divisor is one.

For example, the integers 6, -9, and 14 are relatively prime.

<u>Theorem</u> 8.2

If a and b are relatively prime and $x = x_0$, $y = y_0$ is a solution of the equation $a.x+b.y = n$, then all the solutions of the equation are given by the equations:

$$x = x_0 + k . b \quad \text{and} \quad y = y_0 - k . a$$

for all integral values of k.

8.2.2.2 <u>A Method for Finding a Solution of</u> a.x+b.y=n.

The following method for finding the solution of the Diophantine equation is taken from the book by Griffin

(GRI54).

If GCD(a,b) = d and MOD(n,d) = 0 then by dividing each term of the equation a.x+b.y=n by d, we obtain an equivalent equation, that is, one which is satisfied by all and only the solutions of the original equation. Consequently, it will be sufficient to solve the equation a.x+b.y=n when GCD(a,b) = 1.

We are actually interested in finding one solution of the equation a.x+b.y=n. Once a solution is obtained, Theorem 8.2 can be applied to get all the solutions of the equation.

Case(1). If MOD(n,a) = 0 then $n = n_0 . a$, therefore

$y_0 = 0$ and $x_0 = n_0$ is a solution to the equation.

Case(2). If MOD(n,a) is not zero, then $a \neq \pm 1$ and we may suppose that $1 < |a| < |b|$. Then there exists integers q_1, r_1, q_2 and r_2 such that

$$b = q_1 . a + r_1 \quad \text{for } 0 < r_1 < |a|$$

and

$$n = q_2 . a + r_2 \quad \text{for } 0 < r_2 < |a|$$

Therefore,

$$a . x + (q_1 .a+r_1).y = q_2 .a + r_2$$

Since a solution of a.x+b.y=n exists, $r_1.y-r_2$ is a multiple of a; that is, $r_1.y+a.z = r_2$.

If MOD(r_2,r_1) = 0 then z = 0 and $y = r_2 / r_1$ is a solution to the equation $r_1.y+a.z=r_2$. Then by substituing in the original equation, a value of x for the corresponding y can found.

But if MOD(r_2,r_1) is not zero, then proceed as before, using y and z as the variables and r_1 as the divisor since $r_1 < |a|$. Therefore, we can find integers q_3, r_3, q_4 and r_4 such that

$$a = q_3 . r_1 + r_3 \qquad \text{for } 0 < r_3 < r_1$$

and $r_2 = q_4 . r_1 + r_4$ for $0 < r_4 < r_1$

Therefore,

$$r_1 . y = q_4 . r_1 + r_4 - (q_3 . r_1 + r_3).z$$

and as above,

$$r_3 . z + r_1 . w = r_4 .$$

If we continue in this manner, we find that

AD-A063 900 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN--ETC(U)
SEP 78 S K SHASTRY N00014-76-C-0416
UNCLASSIFIED 78-01 NL

4 OF 6
AD
A063900

$|a| > r_1 > r_3 > \ldots > r_{2s-1} > 0$. We can see that each new set of remainders has an upper bound r_{2k-1} which is smaller than the upper bound of the preceeding set of two. Within a finite number of steps the process will necessarily end, for some r_{2k-1} will divide r_{2k} . Supposing that this happens when $k = s$, the equation $r_{2s-1}.u + r_{2s-3}.v = r_{2s}$ has the solution $v = 0$ and $u = m$; where $r_{2s} = m \cdot r_{2s-1}$. The original variables can then be determined by substitution.

8.2.2.3 Map Vector.

Using the above method of solving Diophantine equation, we can establish a type-8 relation between two nodes of a directed graph. Actually, we are interested in more than just the type-8 relation. We would like to test if the graph is "schedulable" (further discussed in Chapter 9). We associate with each type-8 edge a vector, which will be called Map vector (M_{ij}), defined as follows:

$M_{ij}(k) = p + 1$, if there exists an integer p such that: $1_i + (k-1).c_i = 1_j + p.c_j$;

for all k in the range: $1 <= k <= 1+\lfloor (u_i - l_i)/c_i \rfloor$

and p in the range:

$$0 <= p <= \lfloor (u_j - l_j)/c_j \rfloor.$$

= 0, otherwise.

The weight will also be represented by a 6-tuple: $(x^l_{ij}, x^m_{ij}, c^x_{ij}, y^l_{ij}, y^m_{ij}, c^y_{ij})$ which defines M_{ij} as follows:

$$M_{ij}(x^l_{ij}) = y^l_{ij},$$

$$M_{ij}(x^l_{ij} + c^x_{ij}) = y^l_{ij} + c^y_{ij},$$

$$M_{ij}(x^l_{ij} + 2.\ c^x_{ij}) = y^l_{ij} + 2\ .\ c^y_{ij},$$

...

and $M_{ij}(x^m_{ij}) = y^m_{ij}$.

The elements x^l_{ij}, x^m_{ij}, c^x_{ij}, y^l_{ij}, y^m_{ij} and c^y_{ij} are obtained from the solution of the Diophantine equation which is described in the next sub-section.

To avoid confusion in the representation of a weight vector, the above 6-tuple representation will be replaced by the following representation:

$$M_{ij} = [x^{l}_{ij}, x^{m}_{ij}, c^{x}_{ij}], [y^{l}_{ij}, y^{m}_{ij}, c^{y}_{ij}]$$

As an example, if map vector is given by:

$$M_{ij} = (0,1,3,5,7,0)$$

then, the 6-tuple representation for the above set will be

$$M_{ij} = [2,5,1],[1,7,2]$$

Again, if there is no confusion, the subscripts in the general representation will be dropped as given below:

$$M_{ij} = [x^{l}, x^{m}, c^{x}], [y^{l}, y^{m}, c^{y}]$$

The map vector and its significance in the consistency analysis of a cycle in a directed graph is described in the next chapter.

8.2.2.4 <u>Computation of Map Vector</u>.

Let (x^{0}, y^{0}) be one solution to the Diophantine equation (4). Then all the solutions of the equation are of the form:

$$x = x^{0} - c_{j} . k \qquad -(5)$$

and $$y = y^{0} - c_{i} . k \qquad -(6)$$

for all integral values of k.

However, we want only those values of k which satisfy the following relations:

$$x >= 0 \qquad -(7)$$

$$y >= 0 \qquad -(8)$$

$$l_i + x . c_i <= u_i \quad \text{if } c_i > 0 \qquad -(9)$$

$$l_i + x . c_i >= u_i \quad \text{if } c_i < 0 \qquad -(9)'$$

$$l_j + y . c_j <= u_j \quad \text{if } c_j > 0 \qquad -(10)$$

and $$l_j + y . c_j >= u_j \quad \text{if } c_j < 0 \qquad -(10)'$$

The range of k can be determined from the equations (7) thru (10) and is given for 4 different cases in the following:

Let P, Q, R, and S be defined by:

$$P = (l_i - u_i + x^0 * c_i) / c_i * c_j$$

$$Q = (l_j - u_j + y^0 * c_j) / c_i * c_j$$

$$R = x^0 / c_j$$

$$S = y^0 / c_i$$

And, if the lower bound and upper bound of the range k are given by k'_1 and k'_2 respectively, then k_1 and k_2 are defined by the following equations in four different cases:

Case(1): if $c_i > 0$ and $c_j > 0$ then

$$\begin{cases} k_1 = \mathrm{MAX}(\mathrm{CEIL}(P),\mathrm{CEIL}(Q)) & -(11) \\ k_2 = \mathrm{MIN}(\mathrm{FLOOR}(R),\mathrm{FLOOR}(S)) & -(12) \end{cases}$$

Case(2): if $c_i < 0$ and $c_j < 0$ then

$$\begin{cases} k_1 = \mathrm{MAX}(\mathrm{CEIL}(R),\mathrm{CEIL}(S)) & -(13) \\ k_2 = \mathrm{MIN}(\mathrm{FLOOR}(P),\mathrm{FLOOR}(Q)) & -(14) \end{cases}$$

Case(3): if $c_i < 0$ and $c_j > 0$ then

$$\begin{cases} k_1 = \mathrm{MAX}(\mathrm{CEIL}(S),\mathrm{CEIL}(P)) & -(15) \\ k_2 = \mathrm{MIN}(\mathrm{FLOOR}(R),\mathrm{FLOOR}(Q)) & -(16) \end{cases}$$

and finally,

Case(4): if $c_i > 0$ and $c_j < 0$ then

$$\begin{cases} k_1 = \mathrm{MAX}(\mathrm{CEIL}(R),\mathrm{CEIL}(Q)) & -(17) \\ k_2 = \mathrm{MIN}(\mathrm{FLOOR}(S),\mathrm{FLOOR}(P)) & -(18) \end{cases}$$

Now, if the map vector M_{ij} is given by:

$$M_{ij} = [x^{l}_{ij}, x^{m}_{ij}, c^{x}_{ij}], [y^{l}_{ij}, y^{m}_{ij}, c^{y}_{ij}]$$

then,

$$\text{if } c_j < 0 \text{ then } \left.\begin{aligned} x^{l}_{ij} &= x^{0} - c_j \,.\, k_1 + 1 \\ \& \; x^{m}_{ij} &= x^{0} - c_j \,.\, k_2 + 1 \end{aligned}\right\} \quad -(19)$$

$$\text{otherwise, } \left.\begin{aligned} x^{m}_{ij} &= x^{0} - c_j \,.\, k_1 + 1 \\ \& \; x^{l}_{ij} &= x^{0} - c_j \,.\, k_2 + 1 \end{aligned}\right\} \quad -(20)$$

Also, if $c_i.c_j.k_2 < 0$ then

$$\left.\begin{aligned} y^{l}_{ij} &= y^{0} - c_i \,.\, k_2 + 1 \\ \& \; y^{m}_{ij} &= y^{0} - c_i \,.\, k_1 + 1 \end{aligned}\right\} \quad -(21)$$

Otherwise,

$$\left.\begin{aligned} y^{m}_{ij} &= y^{0} - c_i \,.\, k_2 + 1 \\ \& \; y^{l}_{ij} &= y^{0} - c_i \,.\, k_1 + 1 \end{aligned}\right\} \quad -(22)$$

Also, if $k_1 = k_2$, then $c^x_{ij} = c^y_{ij} = 1$ -(23)

Otherwise, $c^x_{ij} = (x^m_{ij} - x^1_{ij}) / k_2 - k_1$ -(24)

and $c^y_{ij} = (y^m_{ij} - y^1_{ij}) / k_2 - k_1$ -(25)

8.2.2.5 <u>The CHKCONT Procedure</u> (<u>Check Containment</u>).

The function of the CHKCONT procedure is to check for a solution of the Diophantine equation that represents the intersection of the sets of elements that are represented by two nodes of a variable name. If a solution exists, the routine establishes a type-8 relation between the two nodes and associates a map vector with it. The algorithm A8.13 describes the functions of the routine CHKCONT.

<u>Algorithm</u>: A8.13, <u>CHKCONT</u>(I1,J1).

Function: Computes the map vector asociated with the type-8 edge between the two nodes I1 and J1.

Calls: DIOPH.

Called by: SETTYPE.

Step0: If the dimension of the variable name is greater than one, then goto step 6.

Step1: Using the routine DIOPH, check if there exists any solution for the Diophantine equation (4) given in subsection 8.2.2.1. If there is none, then return.

Step2: If there is one solution and that solution is m0 and n0, then see if any of the following two relations are satisfied:

$$m0 < (u_i - l_i) / c_i$$

$$n0 < (u_j - l_j) / c_j$$

If any one of the above relations is satisfied, then no solution exists; return.

Step3: Compute the elements of the map vector associated with the type-8 edge using the equations (19) thru (25) in sub-section 8.2.1.4.

Step4: Set type-8 relation between the nodes I1 and J1 using the routine ADJMSET. Also, save the map vector obtained in step 3.

Step5: Return.

Step6: Using the routine DIOPH, check if there exists a solution for each Diophantine equation in the set of equations (26) given in section 8.2.3. If a solution does not exist for any equation, then return.

Step7: For k = 1 to d perform steps 8 thru 9.

Step8: If m0(k) and n0(k) is a pair of solutions to the k th Diophantine equation in the set of equations (26), then check to see if these solutions satisfy the following two relations:

$$m0(k) < (u_i^k - l_i^k) / c_i^k$$

$$n0(k) < (u_j^k - l_j^k) / c_j^k$$

If any of the above relation is satisfied, then no solution exists, so, return.

Step9: Continue.

Step10: Compute the map vector associated with each dimension of the type-8 edge using equations (19) thru (25) of subsection 8.2.1.4.

Step11: Set the type-8 relation using the routine ADJMSET. Also, save the weighted threshold obtained in step 10.

Step12: Return.

8.2.2.6 The DIOPH Procedure.

The function DIOPH described in the program P8.1, is an implementation of the method of solving a Diophantine equation outlined in sub-section 8.2.1.1. It uses the function GCD to obtain the greatest common divisor of two numbers. It also uses a recursive procedure DIOPHL which computes the series of quotients and remainders and then obtains a solution for the equation.

8.2.2.7 An Illustrative Example.

Let $(l_i, u_i, c_i) = (10,2,-2)$

and $(l_j, u_j, c_j) = (1,10,3)$.

Then the Diophantine equation is given by:

$$10 - 2 . x = 1 + 3 . y$$

i.e., $-2.x - 3 . y = -9$

Hence, $-2.x - (2+1). y = -(2 . 4 + 1)$

i.e., $-y + 2 . s = -1$

Hence, $-y + (1+1).s = -1$

i.e., $s + t = -1$

Therefore, $t = 0$; $s = -1$; $y = -1$; and $x = 6$.

Therefore, (6,-1) is one solution to the above Diophantine equation, and the general solution is given by:

$$x = 6 - 3.k$$

and $y = -1 + 2.k.$

Using equations (24) and (25), we get:

$P = (10-2+6*-2)/-6$

$= 4 /6.$

$Q = (1-10+ -1*3)/-6$

$= 2.$

$R = 6/3.$

$S = 1/2.$

$k_1 = MAX(1,1) = 1.$

and $k_2 = MIN(2,2) = 2.$

Now, using equation (20), we get:

$x^m_{ij} = 6 - 1.3 + 1 = 4$

and $x^1_{ij} = 6 - 2.3 + 1 = 1.$

Now, using equation (21), we get:

$y^1_{ij} = -1 + 2.2 + 1 = 4$

and $y^m_{ij} = -1 + 2 . 1 + 1 = 2.$

And, finally, using equations (24) and (25), we get:

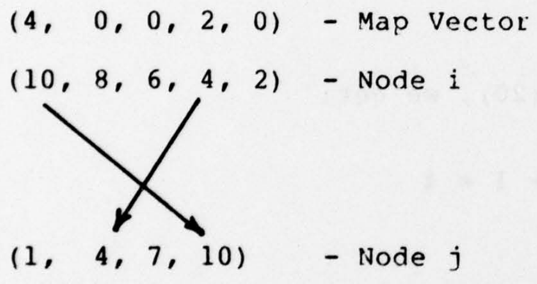
$$c_{ij}^{x} = 3/1 = 3$$

and $c_{ij}^{y} = -2/1 = -2.$

Therefore, the map vector between the nodes i and j is given by:

$$M_{ij} = [1,4,3],[4,2,-2] .$$

Pictorially, the above two nodes can be represented as follows:



The arrows from node i to node j indicate the mappings between the elements of the two nodes. If p th element of node i is mapped into the q th element of node j, then the p th element of the map vector is equal to q. However, if the p th element of node i is not in the set represented by the node j, then the corresponding element of the map vector is zero.

Program: P8.1.

```
DIOPH: PROC(A,B,N,X0,Y0) RETURNS(BIT(1));
/* FUNCTION: OBTAINS ONE SOLUTION TO THE DIOPHANTINE
   EQUATION:
            A * X + B * Y = N
   IF THERE EXISTS A SOLUTION, THE VALUES ARE RETUREED IN
   X0 AND Y0.  IF NO SOLUTION, '0'B IS RETURNED */
 DCL (A,X,N,X0,Y0,D) BIN FIXED;
     D = GCD(A,B); /*GET THE GREATEST COMMON DIVISOR*/
     IF D > 1 THEN DO;
         IF MOD(N,D) = 0 THEN /* DOES NOT SATISFY
                               THEOREM 1.*/
             RETURN('0'B);
         /*SCALE DOWN THE EQUATION*/
         A = A /D; B = B / D; N = N /D;
         END;
     IF MOD(N,A) = 0 THEN DO;
         X0 = N / A; /*SOLUTION IS FOUND*/
         Y0 = 0; RETURN('1'B);
         END;
     IF ABS(A) > ABS(B) THEN DO;
         CALL DIOPHL(B,A,N,Y0);
         X0 = (N-B*Y0)/A;
         END;
```

```
        ELSE DO;
            CALL DIOPHL(A,B,N,X0);
            Y0 = (N-A*X0)/B;
            END;
        RETURN('1'B);
DIOPHL: PROC(A,B,N,X0) RECURSIVE;
    DCL (A,B,N,X0,Q1,Q2,R1,R2) BIN FIXED;
        /*IT IS ASSUMED THAT |A| > |B| */
        Q1 = B / A;
        R1 = B - Q1 * A;
        Q2 = N / A;
        R2 = N - Q2 * A;
        IF MOD(R2,R1) = 0 THEN DO;
            /*FOUND A SOLUTION*/
            X0 = R2/R1; RETURN;
            END;
        CALL DIOPHL(R1,A,R2,X0);
        X0 = (N - B * X0) / A;
        RETURN;
   END DIOPHL;
GCD: PROC(A,B) RETURNS(BIN FIXED);
    DCL (M,N,A,B,R) BIN FIXED;
    /*RETURNS GREATEST COMMON DIVISOR OF A AND B */
        M = A; N = B;
```

```
LO:     R = MOD(M,N);
        IF R = 0 THEN RETURN(N);
        M = N; N = R; GOTO LO;
END GCD;
END DIOPH;
```

8.2.2.8 Variables of Dimension > 1.

In the previous sections, only the variables of dimension one were considered. If a variable is of dimension d, where d is greater than 1, and if a node i in the directed graph represents a subscripted use of that variable, then the notation:

$$A[l_i^1, u_i^1, c_i^1][l_i^2, u_i^2, c_i^2] \ldots [l_i^d, u_i^d, c_i^d]$$

will be used to represent the set of elements contained in node i. A type-8 relationship between two such nodes can be established if there exists at least one solution for each of the following Diophantine equations:

$$\left.\begin{array}{l} m_i^1 . c_i^1 - n_j^1 . c_j^1 = l_j^1 - l_i^1 \\ m_i^2 . c_i^2 - n_j^2 . c_j^2 = l_j^2 - l_i^2 \\ \quad \cdots\cdots \end{array}\right\} \text{----(26)}$$

$$\left.\begin{array}{l} \vdots \\ m^{d}_{i} \cdot c^{d}_{i} - n^{d}_{j} \cdot c^{d}_{j} = l^{d}_{j} - l^{d}_{i} \end{array}\right\}$$

where $m^{1}_{i}, m^{2}_{i}, \ldots, m^{d}_{i}$ and $n^{1}_{j}, n^{2}_{j}, \ldots, n^{d}_{j}$ are the variables in the equations.

The solution to each of the above diophantine equations can be denoted by the 6-tuple Map vector described in section 8.2.1.3. However, to extend the notation to a general case, the map vector corresponding to the k th dimension will be denoted by:

$$[xl^{k}, xm^{k}, cx^{k}, yl^{k}, ym^{k}, cy^{k}]$$

As an example, the map vector associated with the type-8 edge between the nodes 2 and 3 in Figure 8.6 is given by:

[1,4,1,1,4,1] [1,3,1,2,4,1]

and that between nodes 5 and 2 is given by:

[1,3,1,2,4,1] [1,1,1,1,1,1]

8.2.2.9 <u>A Special Case of Subscripted Variables.</u>

The general representation of a subscripted variable node which is given by:

$$A[l^1_i, u^1_i, c^1_i][l^2_i, u^2_i, c^2_i] \ldots [l^d_i, u^d_i, c^d_i]$$

for a variable of dimension works very well when an assertion maps a set of elements of one dimension of a source variable to a set of elements in the same or another dimension of the target variable. However, in some cases, a set of elements of one dimension in a source variable may be mapped to a set of elements distributed among more than one dimension of the target variable. Following assertion is an example of such a case:

```
    FOR I = 2 TO 4 A(I,I) = A(I,1)
```

In the above assertion, the target variable represents the elements A(2,2), A(3,3) and A(4,4) which cannot be represented using the general form above. However, in such cases, we can merge the two dimensions into one dimension. For example, if the dimension of the variable is 2 and both dimensions use the same subscript name, as in the avove assertion, then the merged representation of the variable is given by:

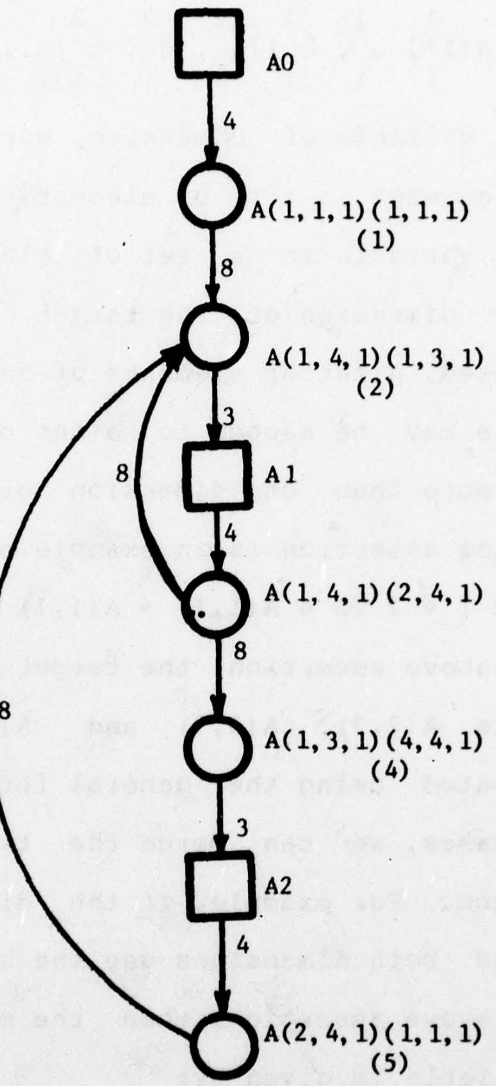


Figure 8.6. Directed Graph for the Assertions:

```
A0:  A(1,1) = 1;
A1:  FOR I = 1 TO 4 & J = 2 TO 4
     A(I,J) = A(I,J-1)*2;
A2:  FOR I = 2 TO 4 A(1,I) = A(I-1,4);
```

$$A[l_i^{1'}, u_i^{1'}, c_i^{1'}]$$

where:

$$l_i^{1'} = (l_i^{1} - 1) . x + l_i^{2}$$

$$u_i^{1'} = (u_i^{1} - 1) . x + u_i^{2}$$

$$c_i^{1'} = c_i^{1} . x + c_j$$

and x is the maximum number of elements in the first dimension of the variable.

For example, the variable name A(I,I) will be represented by A[6,16,5] when the variable size = 4x4 and I varies from 2 to 4, as in the above assertion. Now the CHKCONT procedure can be extended to check the intersection of a normal representation of a variable name with another merged representation. This extension will be illustrated using the example in the directed graph of Figure 8.7.

If the node (2) was represented normally, then the Diophantine equations 1 and 2 in the set of equations (26) would have been used to check the containment. The same concept is extended by using the Diophantine equation with more than one variable. In other words, there exists a

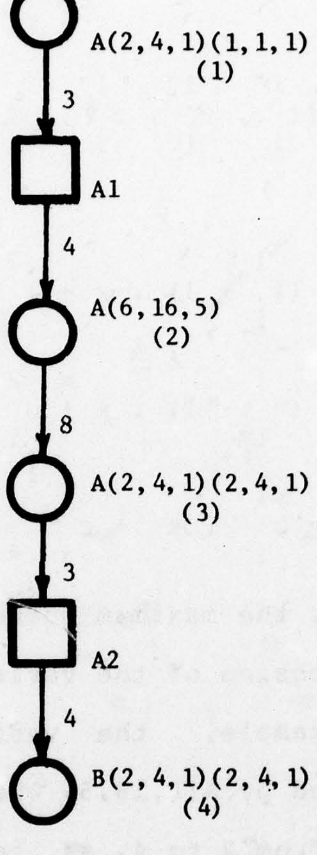


Figure 8.7. Directed Graph for the Assertions:

A1: FOR I = 2 TO 4 A(I,I) = A(1,I);

A2: FOR I = 2 TO 4 & J = 2 TO 4

B(I,J) = A(I,J);

type-8 edge between the nodes (2) and (3) if there exists a solution to the Diophantine equation:

$$6 + 5 . x = (2-1 + y.1).a + 2 + z.1$$

where a is the size of the last dimension of A, which is 4 in the present example. The above equation is a Diophantine equation in 3 variables x, y and z, and can be rewritten as:

$$5.x - 4.y - z = 0$$

In general, if two dimensions d1 and d2 (such that d2 > d1) of a variable node i use the same subscript name, and all the subscript names corresponding to the different dimensions of node j are unique, and the node i is a type-8 predecessor of node j, then the Diophantine equation:

$$l_i^{d1'} + c_i^{d1'} . x = (l_j^{d1} -1 + c_j^{d1} . y).a_{d1d2} +l_j^{d2} +c_j^{d2} .z$$

will be used to establish type-8 relation between the nodes i and j.

a_{d1d2} used in the above equation is a constant defined as follows:

$$a_{d1d2} = 1 \text{ if } d1=d2,$$

$$= s_{d3} . a_{d1d3} \quad \text{for all} \quad d1 < d3 <= d2$$

where s_{d3} is the size of the d3 th dimension of the variable.

The same concept can be applied if more than two dimensions of a variable represented by the node i use the same subscript name; and also, if such a node exists in the head and/or tail of a type-8 edge. A solution of a Diophantine equation of 3 variables can be obtained using standard methods.

There is an alternate method to find the type-8 relationship between the special nodes described above. This method is sketched in the following steps.

Step1: Let i be a target variable node, all the dimensions of which use unique subscript names.

Step2: Let j be a source variable node, the p th and q th dimensions of which use the same subscript name. Also, assume that $p < q$.

Step3: There exists a type-8 relation between i and j if there exists a solution for the Diophantine equation:

$$l_i + c_i \cdot x = l_j + c_j \cdot y \qquad -(27)$$

where -

$$l_i = (l_i^p - 1) \cdot S_i^q + l_i^q$$

$$u_i = (u_i^p - 1) \cdot S_i^q + u_i^q$$

$$c_i = c_i^q$$

$$l_j = (l_j^p - 1) \cdot S_i^q + l_j^q$$

$$u_j = (u_j^p - 1) \cdot S_i^q + u_j^q$$

$$c_j = c_j^p \cdot S_i^q + c_j^q$$

and

$$S_i^q = u_i^q - l_i^q + c_i^q$$

As an example, consider the node i (denoted by: A(1,5,1)(1,5,2)) representing the set of elements:

A(I,J)   for I = 1 to 5 & J = 1 to 5 by 2;

and the node j (denoted by A(2,4,1)(2,4,1)) representing the set of elements:

A(I,I)   for I = 2 to 4.

Assuming that A is of size 5x6, we can see that the node i represents the underlined elements in the following matrix:

$$\begin{array}{cccccc}
\underline{1} & 2 & \underline{3} & 4 & \underline{5} & 6 \\
\underline{7} & 8 & \underline{9} & 10 & \underline{11} & 12 \\
\underline{13} & 14 & \underline{15} & 16 & \underline{17} & 18 \\
\underline{19} & 20 & \underline{21} & 22 & \underline{23} & 24 \\
\underline{25} & 26 & \underline{27} & 28 & \underline{29} & 30
\end{array}$$

whereas, the node j represents the underlying elements in the following matrix:

$$\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
7 & \underline{8} & 9 & 10 & 11 & 12 \\
13 & 14 & \underline{15} & 16 & 17 & 18 \\
19 & 20 & 21 & \underline{22} & 23 & 24 \\
25 & 26 & 27 & 28 & \underline{29} & 30
\end{array}$$

Using the set of equations (28), we get:

$$S_i^q = 5 - 1 + 2 = 6$$

$$l_i = 0 . 6 + 1 = 1$$

$$u_i = 4 . 6 + 5 = 29$$

$$c_i = 2$$

$$l_j = 1 . 6 + 2 = 8$$

$$u_j = 3 . 6 + 4 = 22$$

and $c_j = 1 . 6 + 1$

And the Diophantine equation (27) becomes:

$$1 + 2.x = 8 + 7.y$$

Using the method of solution described earlier, we can get the map vector between the two nodes, and is given by:

$$W_{ij} = [8,15,1,2,4,2]$$

Note that S_i^q used in the set of equations (28) is a fictious size of the q th dimension of the variable. This size is used to obtain a linear transformation of the elements represented by the two dimensions p and q of the variable into one dimensional array. That is why we were in a position to use the Diophantine equation of only two variables to derive the type-8 relationship.

CHAPTER 9

THE COMPLETENESS AND CONSISTENCY ANALYSIS

In this chapter, the completeness and consistency analysis of MODEL specification will be described. An overview of the completeness and consistency analysis phase of the MODEL processor is shown in Figure 9.1. The completeness criteria consists essentially of the requirement that the directed graph (that represents the MODEL specification) is well-formed in the following sense:

(1). Each row and column in the precedence matrix must have at least one entry of a precedence relationship with the following exceptions: (a) Rows representing the source media statements may not have any entry and the columns representing target media statements may not have any entry. (b) Rows representing source field names may not have any entry (when the field is never used to define any other varaible).

(2). If a node represents a set of elements of an array and that node is used as source of an assertion, then the union of the set of elements represented by its predecessor nodes must be the same as (or a superset of) the set of elements represented by the node.

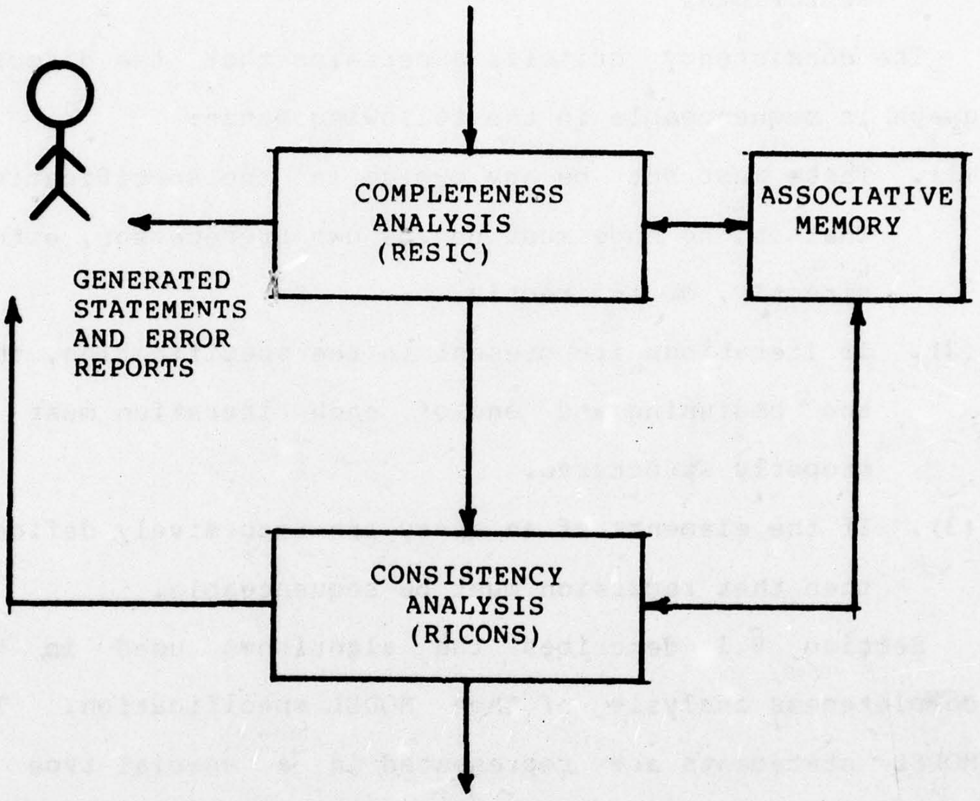

COMPLETENESS ANALYSIS (RESIC)
ASSOCIATIVE MEMORY
GENERATED STATEMENTS AND ERROR REPORTS
CONSISTENCY ANALYSIS (RICONS)

Figure 9.1. Overview of Phase-3 of MODEL Processor.

(3). Each repeating data name must be subscripted by an appropriate number (equal to the dimension) of subscripts.

The consistency criteria ascertains that the directed graph is sequenceable in the following sense:

(1). There must not be any cycles in the specification. That is, no node must be its own predecessor, either directly, or indirectly.

(2). If iterations are present in the specification, then the beginning and end of each iteration must be properly structured.

(3). If the elements of an array are recursively defined, then that recursion must be sequenceable.

Section 9.1 describes the algorithms used in the completeness analysis of then MODEL specification. The MODEL statements are represented in a special type of directed graph, called array graph, for ease of analysis. Section 9.2 describes the concept of array graphs and the results on the sequenceability of array graphs. Section 9.3 describes the algorithms used in the consistency analysis of the MODEL specification. Also, Figures 9.2 and 9.3 give the functional overview of the completeness and consistency analysis procedures respectively. Some examples are described in section 9.4.

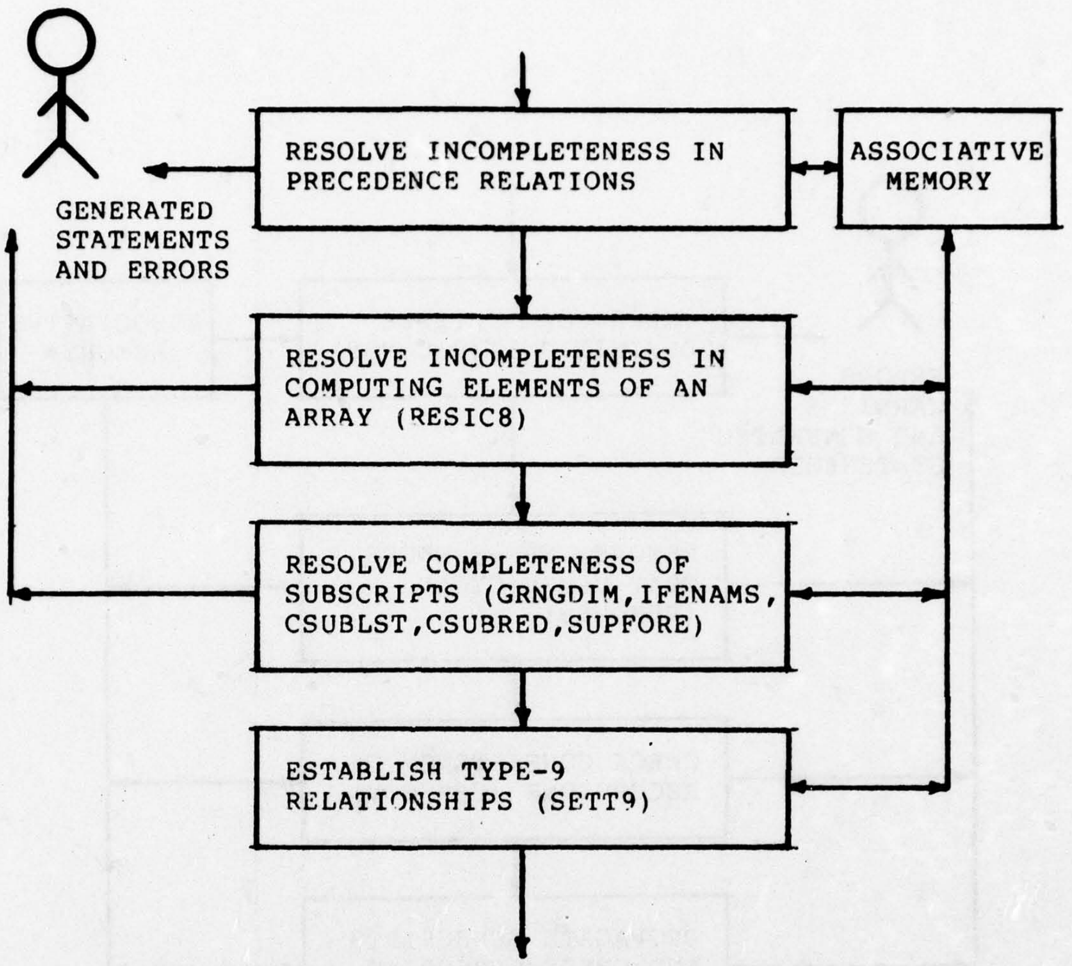

RESOLVE INCOMPLETENESS IN PRECEDENCE RELATIONS
ASSOCIATIVE MEMORY
GENERATED STATEMENTS AND ERRORS
RESOLVE INCOMPLETENESS IN COMPUTING ELEMENTS OF AN ARRAY (RESIC8)
RESOLVE COMPLETENESS OF SUBSCRIPTS (GRNGDIM,IFENAMS, CSUBLST,CSUBRED,SUPFORE)
ESTABLISH TYPE-9 RELATIONSHIPS (SETT9)

Figure 9.2. Overview of Completeness Analysis. (Procedure RESIC).

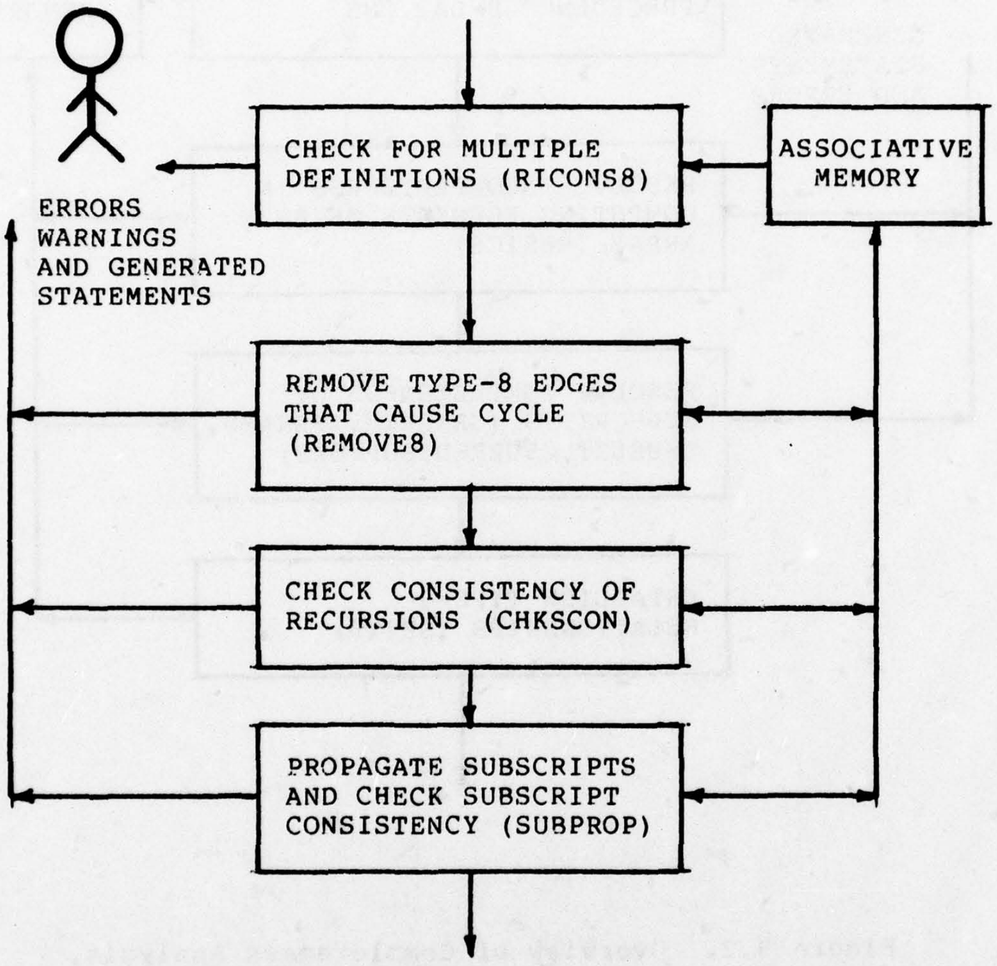

ERRORS
WARNINGS
AND GENERATED
STATEMENTS
CHECK FOR MULTIPLE DEFINITIONS (RICONS8)
ASSOCIATIVE MEMORY
REMOVE TYPE-8 EDGES THAT CAUSE CYCLE (REMOVE8)
CHECK CONSISTENCY OF RECURSIONS (CHKSCON)
PROPAGATE SUBSCRIPTS AND CHECK SUBSCRIPT CONSISTENCY (SUBPROP)

Figure 9.3. Overview of Consistency Analysis. (Procedure RICONS).

9.1 Completeness Analysis.

This section describes the algorithms used in the completeness analysis of MODEL specification. As mentioned in the beginning of this chapter, completeness analysis can be divided into three parts. The first part checks for the completeness of the precedence matrix (Procedure RESIC), and the second part checks that every element of an output array is defined (Procedure RESIC8). In both cases, the incompleteness is resolved by generating appropriate assertions or data description statements.

The third part checks for completeness in subscript specification. If a data name is repeating, but no subscript (or not enough subscript) is specified, then default subscripts ('*') are supplied.

In the following, the above three parts are described in greater detail.

9.1.1 Incompleteness in Precedence Relations.

For a directed graph to be complete, it must be well formed in that there must be an entry in each row and column of the precedence matrix, except in the following special cases:

(1). A column that represents an "input node" need not contain any entry. The input nodes are those nodes that represent (a) source media statements, and (b) assertions that define variable names by constants.

(2). A row that represents an "output node" need not contain any entry. In MODEL, the nodes that represent target media statements are the only output nodes.

(3). A row that represents a source field name (or an interim data name) need not have any entry. In this case, the particular name is never used to define any other name.

If the above completeness criteria are not satisfied, an appropriate data description statement or an assertion is generated according to the following rules:

(1). If the node under consideration represents a file, record, group or field data name, and there does not exist a parent for that data name (that is, if the column or row of that data name does not contain any entry according as that data name is a source data name or target data name respectively), then a corresponding data description statement is generated as given by the following table:

WHEN THE DATA NAME IS A	THE GENERATED STATEMENT IS
SOURCE FIELD	$INREC IS RECORD;
SOURCE GROUP	$INREC IS RECORD;
TARGET FIELD	$OUTREC IS GROUP;
TARGET GROUP	$OUTREC IS GROUP;
SOURCE RECORD	SYSIN IS FILE;
TARGET RECORD	SYSPRINT IS FILE;
FILE (SOURCE OR TARGET)	$DISK IS MEDIA

Also, the data name in the generated statement is assumed to be the parent of the data name under consideration. The precedence matrix is also updated to include the parent-daughter relationship resulting from the generated statement.

(2). If the node under consideration represents an interim data name or a target field name, and if there does not exist any entry in the column of that node, then an assertion is generated according to one of the following rules:

(a). If there exists a similarly named source or interim data name, then that name is used to define the data name using an assertion. For example, if the node under consideration represents the field name "REPORT.TRANS#", and there exists another field name "TRANS#" in a source file "TRANSACTION", then the assertion

```
REPORT.TRANS# = TRANSACTION.TRANS#;
```

is generated.

(b). If the incompleteness could not be resolved by (a) above, then one of the following two assertions is generated according as the data name is defined to be of type arithmetic, or character respectively:

```
<datname> = 0;

<data-name> = '';
```

There are two exceptions to the rule (a) above. These are:

(i). If the data name is "FILLER", then the rule (b) is used even though there may be a similar source data name.

(ii). If the dimension of the data name under consideration is smaller than the similarly named source data name, then the reduction function ANY is used to select a particular element of source data name. For example, for the following set of statements:

```
TRANS IS FILE;
  PRODUCT IS RECORD(TRANS,(*));
    TRANSACTION IS GROUP(PRODUCT,(*));
      PRODUCT_NUMBER IS FIELD(TRANSACTION);
...
```

```
REPORT IS FILE;
  REP1 IS RECORD(REPORT,(*));
    PRODUCT_NUMBER IS FIELD(REP1);
```

the following assertion is generated:

```
REP1.PRODUCT_NUMBER = ANY(TRANS.PRODUCT_NUMBER);
```

The procedures GIMPA and GIMPI are used to perform the functions of rules (a) and (b) respectively.

9.1.2 Incompleteness in Computing Elements of an Array.

If an array is defined as a target field or interim, then there must exist an assertion to define every element of that array. If not, the specification is incomplete. Because, the nodes of our directed graph represent a set of elements of an array, such incompleteness cannot be detected by analyzing the entries in the precedence matrix. Therefore, some other procedure must be adopted. The Procedure RESIC8 is used for this purpose. The functions of this procedure is summarized in the following:

Step1. Obtain all the type-8 predecessors for each node in the directed graph (using the routine OBT#P).

Step2: For each node, i, in the graph, perform steps (3) thru (4).

Step3: Find the union, U, of the set of elements represented by the type-8 predecessors of node i.

Subtract U from the set represented by the node i (using routine RANGSR).

Step4: If the resulting set, R, obtained in step3 is not empty, then those elements in R are not defined by any assertion. Therefore using the procedure GAFSUBS, generate a default assertion to define every element in R.

The routine OBT#P used in stepl above computes some vectors that are used later on during the consistency analysis phase (see section 9.3.2). Some of these vectors are: #PRED (# of predecessors), #PRED8 (# of type-8 predecessors), #SUCC8 (# of type-8 successors), PREDL (a list of all predecessors), PRED8L (a list of all type-8 predecessors), and SUCC8L (a list of all type-8 successors). The use of these vectors is described in section 9.3.2.

To illustrate the functions of the procedure RESIC8, consider the following set of statements:

```
A(1) = 0;
FOR I = 3 TO 9 BY 2  A(I) = A(I-2);
A IS INTERIM(10);
```

In the above example, the elements: A(2.I), I = 1 to 5, are not defined, and therefore, the following assertion is generated by the system:

```
FOR I = 1 TO 5 A(2*I) = 0;
```

9.1.3 Completeness Analysis of Subscripts.

Basically, the completeness analysis of subscripts consists of the following steps:

(1). Obtain the range and dimensionality of each source, target and interim data names and then propagate them to the assertions that use them (Procedure GRNGDIM).

(2). Identify the "for-each" names (or '*') used in each assertion with one of the supplied subscript names, or generate a new subscript statement for that "for-each" name (Procedure IFENAMS).

(3). Obtain a list of all data names that use a subscript name and obtain a parent for that subscript name from one of the names in the list (Procedure CSUBLST).

(4). Analyze the list obtained in step (3) for redundancy & inconsistency (Procedure CSUBRED).

(5). Supply "for-each" expressions for each assertion that uses repeating data name (Procedure SUPFORE).

The procedures that perform the above functions are described in the following.

9.1.3.1 <u>The Procedure GRNGDIM (Get Range and Dimension)</u>.

This procedure obtains the range and dimensionality of every data name in the dictionary. If the obtained dimension is different from the number of subscripts used for the same name in any assertion, then the assertion (or the data description statement) is modified. For example, consider the following set of statements:

```
        A IS GROUP((*));
              C IS FIELD (A,(50));
        X IS INTERIM ((*,50));
        A1: X(I) = C(I);
```

In this example, the number of subscripts specified for the data names X and C in the assertion is different from the actual dimension (obtained from the corresponding data description statements). Therefore, the assertion A1 is changed to:

```
        A1: X(*,I) = C(*,I);
```

However, if the data description for the interim data name X in the above example was:

```
        X IS INTERIM;
```

Then the procedure GRNGDIM changes the dd statement for X to:

```
        X IS INTERIM((*));
```

and the assertion is changed to:

```
        A1: X(I) = A(*,I);
```

Later on, during the subscript propagation (see the description of the Procedure SUBPROP), the data description of X will again be changed to:

```
        X IS INTERIM ((*,*));
```

and the assertion will be changed to:

```
        A1: X(*,I) = C(*,I);
```

9.1.3.2 <u>The Procedure IFENAMS</u> (<u>Identify</u> "<u>FOR-EACH</u>" <u>Names</u>).

If a repeating data name is not subscripted in an assertion, then the data name in the assertion is changed to include the default subscript name ("for-each" or "*") as described in the earlier sub-section. Alternately, the user might have used those for-each names himself as in the following assertion:

```
        A(*) = B(*) + C(*);
```

The function of the procedure IFENAMS is to supply an appropriate subscript name for each "for-each" name in an assertion. If a "for-each" name is used as a subscript for the k th dimension of a data name <dname>, then the subscript is changed to a subscript name using the following rules:

(1). If a subscript name is already specified for the k th dimension of <dname> (using a subscript statement), then that subscript name is used as the subscript.

(2). If no subscript name is specified for the k th dimension of <dname>, then the assertion in which the <dname> is used is searched for a subscripted data name and an acceptable subscript name. If a subscript name is found, then that subscript name is used as the subscript for the k the dimension of <dname>.

(3). If the search in steps (1) and (2) is unsuccessful, then a new subscript name is supplied. Also, the same name is propagated to the k th dimension of every variable used in the assertion, if "for-each" was specified for that dimension.

For example, if the assertion is of the form:

```
        A1: A(*) = B(*) + C(*);
```

then, the assertion is changed to:

```
        A1: A(FOREACH$A) = B(FOREACH$A) + C(FOREACH$A);
```

and the following subscript statement is generated:

```
        FOREACH$A IS SUBSCRIPT(A);
```

Note that the parent name is arbitrarily chosen among the possible candidates. This parent-daughter association may be changed after further analysis in procedure CSUBLST, which is described in the next section.

If another assertion A2 was also, specified:

```
A2: X(*) = Y(*) * C(*);
```

The for-each names in this assertion is changed to FOREACH$A, because, the for-each name associated with C was changed to FOREACH$A while analyzing the assertion A1. Therefore, the assertion A2 is changed to:

```
A2: X(FOREACH$A) = Y(FOREACH$A) * C(FOREACH$A);
```

As another example, if the assertion is given by:

```
A3: AA(*,*) = BB(*) + CC(*);
```

Then, two subscript names will be supplied, thus changing the assertion to:

```
A3: AA(FOREACH$AA1,FOREACH$AA2) = BB(FOREACH$AA2) +
               CC(FOREACH$AA2);
```

While a subscript is supplied to a data name, the derivation tree of the corresponding assertion (in which the data name is used) must also be updated. This is done in Procedure UPDASSN. The procedure GSUBNPT is used to obtain a new subscript name. Whenever a new subscript statement is generated for a data name, then that data name is assumed to be the parent of the subscript name. However, if the data name is not repeating, then the parent structure of that data name is searched for the repeating data name. For example, for the following set of statements:

```
        A IS GROUP(*);
          B IS FIELD(A,(*));

        C (*,*) = B(*,*);
```

two subscript statements are generated for the two subscripts, and the assertion is changed to:

```
        C(FOREACH$A,FOREACH$B) = B(FOREACH$A,FOREACH$B);
```

Also, the data names A and B are assumed to be the parents of the subscript names FOREACH$A and FOREACH$B respectively. The procedure GEXACTD is used for obtaining the repeating data name.

Also, whenever a subscript statement is generated, the procedure SSUBRNG is called to set the range for the subscript from the corresponding range in the data description statement of its parent repeating data name.

9.1.3.3 <u>The Procedure CSUBLST</u> (<u>Create Subscript List</u>).

This procedure creates a list of all data names that use a particular subscript name. Also, from this list, it obtains a parent name for the subscript name using the following process:

(1). Search for the data name in the list that has the largest dimension. If more than one data name with the largest dimension exists, then search for an arbitrary source data name among them.

(2). If a parent name is not associated with the subscript, or if the specified parent name has a dimension less than that of the data name obtained in step (1), then the parent name is changed to the data name obtained in step (1).

As an example, if the data name list for a subscript name I is given by:

A(I), B(I), X(I).

and X(I) and B(I) are source data names, and I does not have any parent, then either B or X is supplied as the parent name for I.

As another example, if the data name list for a subscript name I is given by:

A(I,J), B(I), C(I).

Then, A is assumed to be the parent of I even if A is a target variable.

9.1.3.4 <u>The Procedure CSUBRED (Check Sub. Redundency)</u>.

This procedure is used to check redundancies and inconsistencies in subscript specification. The check is performed on the list of data names associated with a subscript name (which is obtained using the procedure CSUBLST described in the earlier sub-section). Subscript inconsistency is due to the use of same subscript name for two different source data names or two different target data names. For example, consider the following set of statements:

```
        G IS GROUP;
          A IS FIELD(G,100);
          B IS FIELD(G,100);
        C IS FIELD(X,100);
        C(I) = A(I) + B(I);
```

In this example, the subscript name I is used for the data names A and B, which are not related. This type of inconsistency, in many cases may be acceptable, and therefore, is not treated as error, but a warning is sent to the user for each such inconsistency.

Subscript redundancy is due to the use of different subscript names for the same data name. For example, if the list of data names associated with two subscript names I and J are given by:

```
        A(I), B(I), X(I);
   and  C(J), B(J), Y(J).
```

then the data name B is subscripted by both I and J. Therefore, one of the names can be replaced by the other. If J is changed to I, the subscript statement for J is deleted and the list of data names associated with the data name I is changed to:

A(I), B(I), X(I), C(I), Y(I).

However, there is one exception to the above rule. If two subscript names I and J are used as subscripts for a data name, but the parents of I and J are the same name (I and J being associated with the different dimensions of that data name), then there is no redundancy. For example, if the data name lists associated with I and J are given by:

A(I), B(I,J);

and A(J), B(I,J).

respectively, and if I is associated with the first dimension of B and J is associated with the second dimension of B, then identifying the subscript name I with the name J will violate the second criteria used in the subscript consistency check of procedure ASUBST. Therefore, in this case, I and J are treated as two different subscripts. This type of situation can arise in cases like the following:

```
B(I,J) = X(I) * Y(J);
```

In this case, matrix B is assumed to be the outer product of the vector A.

If two subscript names are found to be equivalent (that is, one of them is redundant), then one of the names is "deleted" from the associative memory. (Actually, the redundant subscript name is not deleted physically. But, a linked list of all equivalent subscript names is maintained in the associative memory. Also, an error code is moved to the data area of the storage entry of every member in the equivalent class, except one, to indicate that that statement is a deleted statement.)

9.1.3.5 The Procedure SUPFORE (Supply For-each Expns.).

This procedure supplies for each expressions for every assertion in which a subscript data name is used, but no range for that subscript is specified. For example, consider the following two statements:

```
        I IS SUBSCRIPT(A,(1,100,1));
        A(I) = B(I) + C(I);
```

In this example, the assertion does not contain any range for the subscript name I. Therefore, the range for I is obtained from the subscript statement for I, and the assertion is changed to:

```
            FOR I = 1 TO 100 BY 1 A(I) = B(I) + C(I);
```

It uses the procedure MERGSUB described in section 8.2.3.3 to supply the for-each expressions.

9.1.4 Some Additional Processing.

Once the completeness is complete, two additional functions are performed by the routine RESIC. These are:

(1). Establishes type-9 relationships between the sister nodes in the tree structure corresponding to the input, output, or interim data names. This function is performed by the routine SETT9.

(2). Obtains the number of different representations for a name. This information is stored in #OFENTRIES of the structure "DATA-LIST" (see section 6.1.4.1). This entry is used in the documentation phase to supply minimum number of qualifications for any data name. The Procedure SET#OFE is used to compute #OFENTRIES.

9.2 Array Graph Representation.

In this section, we will define the concept of an "array graph", which is a generalized representation of a directed graph. In general, an array graph is a compact representation of a conventional directed graph in which each node represents a variable name or an element of a variable name. We are interested in performing the graph algorithms directly on the array graph. In particular, we are interested in obtaining a "schedule"* for the array graph. We will call the array graph sequenceable if the underlying graph (which is equivalent to the original graph, and in which each node represents a single element) is acyclic. We will see that not all array graphs are sequenceable; however, we can decompose a set of nodes of some of those array graphs, and obtain a schedule for the decomposed graphs. Using appropriate definition of decomposability, we will also derive a necessary and sufficient condition for sequenceability. We will also

---

* Informally, schedule is a sequence of two control elements (called, do-element and end-element), and the compound nodes in the graph. The nodes in the sequence are topologically sorted. If a set of nodes are to be iterated, then that set will be preceded by a do-element, and succeded by an end-element in the schedule. A formal definition of schedule is given in section 9.2.3.

call a graph, "schedulable" if we can obtain a schedule for that graph. We will show that not all sequenceable graphs are schedulable, however, we can decompose a sequenceable graph to obtain a schedulable graph. We will describe our graph analysis after introducing some definitions. An overview of the results on the array graph is shown in Figure 9.4.

To make our graph analysis, we will use a simpler representation of our directed graph. This simpler representation consists of only two types of edges: (1) Ø1 edge-connecting a source variable node and a target variable node (this is represented by a type-3 edge, an assertion node, and a type-4 edge in the actual representation described in earlier chapters) and (2) Ø2 edge - connecting a target variable node and a source variable node (this edge is the same as the type-8 edge described earlier).

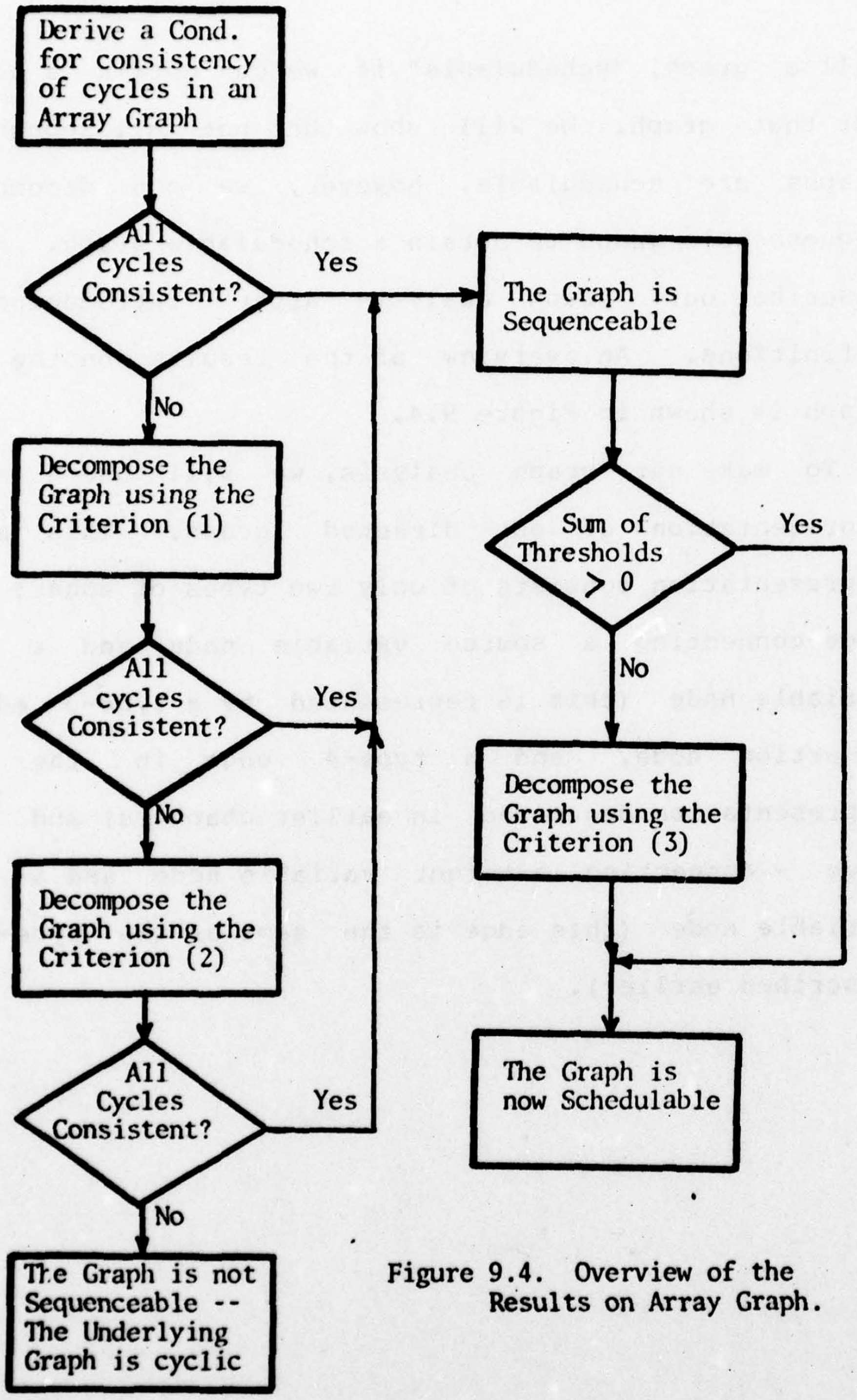

Derive a Cond. for consistency of cycles in an Array Graph
All cycles Consistent?
Yes
No
The Graph is Sequenceable
Decompose the Graph using the Criterion (1)
Sum of Thresholds > 0
Yes
No
All cycles Consistent?
Yes
No
Decompose the Graph using the Criterion (3)
Decompose the Graph using the Criterion (2)
The Graph is now Schedulable
All Cycles Consistent?
Yes
No
The Graph is not Sequenceable -- The Underlying Graph is cyclic

Figure 9.4. Overview of the Results on Array Graph.

9.2.1 Array Graph.

An array graph G is a pair (N,E), where N is a set of compound nodes and E is a set of compound edges. Each compound node has the form:

$$A(l,u,c)$$

where A is the node (variable, or array) name, and l, u and c define the array range. Thus this node represents the set of elements: A(l), A(l+c), ... A(l+k.c), where $k = \lfloor (u-1)/c \rfloor$.

A compound edge will have the form:

$$A(a.I+b) <- B(c.I+d) \quad \text{for } l <= I <= u$$

where a, b, c, d, l and u are integer constants.

For a given array graph G = (N,E), we can define the underlying graph $G_u = (N_u, E_u)$ which is a conventional graph obtained by considering each of the array components as individual nodes. Thus,

$$N_u = \{A(l+I.c) \mid 0 <= I <= \lfloor (u-1)/c \rfloor \text{ and } A(l,u,c) \in N\}$$

and

$$E_u = \{A(a.I+b) <- B(c.I+d) \mid l <= I <= u$$
$$\text{and } (A(a.I+b)<-B(c.I+d),\ l<=I<=u) \in E)\}.$$

We will define an array graph G to be acyclic if its underlying graph G_u is acyclic.

In the above definition, we restricted variable names to be of dimension 1. In a general case, a compound node will be represented by:

$$A(l^1, u^1, c^1)(l^2, u^2, c^2) \ldots (l^n, u^n, c^n)$$

which represents the set of variables:

$$\{A(l^1 + k^1 . c^1, l^2 + k^2 . c^2, \ldots, l^n + k^n . c^n) \mid$$

$$0 <= k^1 <= \lfloor (u^1 - l^1)/c^1 \rfloor ,$$

$$0 <= k^2 <= \lfloor (u^2 - l^2)/c^2 \rfloor ,$$

$$\vdots$$

$$\text{and} \quad 0 <= k^n <= \lfloor (u^n - l^n)/c^n \rfloor \}$$

Where n is the dimension of the variable. The representation of a compound edge can also be extended in a similar manner. In the following, only arrays of dimension 1 are considered. The analysis of general array graphs in which the nodes represent arrays of dimension > 1, is quite involved, and therefore, the general case will not be discussed. However, some results on the analysis of array graphs of dimension 1 can be extended to the general case, and this extension will be described in section 9.2.16.

As mentioned earlier, we are interested in sequencing the array graph, and we can see that an array graph G is sequenceable if the corresponding underlying graph is acyclic.

In general, it is difficult to say if an array graph is acyclic from its initial representation. This is because, the different nodes representing the same data name may have different (intersecting or non-intersecting) ranges. To overcome this problem, we will introduce the concept of an "induced graph" which is defined below.

9.2.2 <u>Induced Array Graph</u>.

For a given array graph G = (N,E), we construct a bipartite induced graph $G_i = (N_i, E_i)$ as follows: The set of nodes $N_i = L \cup R$ is a union of two subsets, left nodes (L nodes) and Right nodes (R nodes). The set of directed $E_i = \Phi_1 \cup \Phi_2$ is also partitioned into Φ_1, a set of edges going from R nodes into L nodes; and Φ_2, a set of edges going from L nodes into R nodes. For each edge in the original graph:

```
   A(a.I+b) <- B(c.I+d)      1 <= I <= u
```

We add to the set L, the node A(a.1+b,a.u+b,a), and to R, the node B(c.1+d,c.u+d,c), and draw a Ø1 edge leading from the latter to the former.

After the construction of L, R & Φ_1 is thus complete, we construct Φ_2 by drawing Ø2 edge between any $l \in L$ and $r \in R$, which have a non-empty intersection (that is, have identical variable names and non-empty range intersection).

Consider for example, the array graph:

```
        A(1) <- C
        A(I) <- A(I-1)       For 2<=I<=4
```

Its induced graph is given in Figure 9.5 (denoting L nodes by Squares and R nodes by Circles).

We do impose another restriction on our induced graphs and that is, no two L nodes have a non-empty intersection. This implies that in the original specification, no two compound edges refer to a common underlying element. If this is not satisfied originally, we can always form intersections of compound edges and break each range into several sub ranges until this requirement is met.

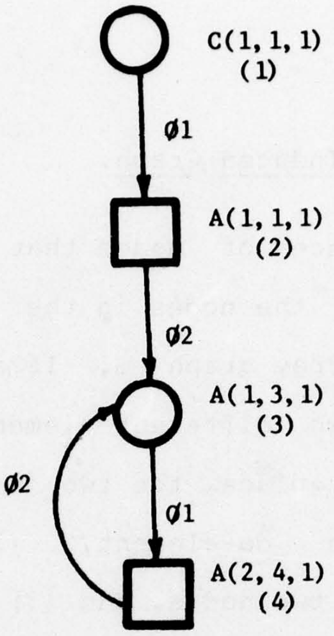


Figure 9.5. Induced Graph for the Assertions:

A(1) = c;
A(I) = A(I-1) For $2 \leq I \leq 4$

9.2.3 Schedule for an Induced Graph.

Let S be a sequence of nodes that represents the topological sorting of the nodes in the underlying graph corresponding to an array graph G. If any two adjacent nodes in S represent two different elements of a variable name, then we will replace the two nodes in S by a sequence of (1) a do-element, (2) a compound representation of the two nodes, and (3) an end-element.

The do-element is of the form:

```
DO I = 1 TO u BY c;
```

The end-element is just the keyword: END;
and the compound node is of the form: A(I).

Thus, if a do-element preceeds a compound node, then the range in the do-element specifies the range of elements that the compound node represents. The end-element is just used to terminate the effect of the do-element immediately preceeding it. For example, if we have the following sequence of elements in S:

```
B(1); A(1); A(3); B(3); ...
```

then, the sequence can be replaced by:

```
B(1);
DO I = 1 TO 3 BY 2;
    A(I);
END;
B(3);
```

We can apply similar reduction process to more than two adjacent elements of a same name in the sequence, or to a sequence of elements of two different names, the elements of which occur alternately in the sequence. For example, the following sequence of nodes:

```
C(1); A(1); B(1); A(2); B(2); C(2); ...
```

can be replaced by:

```
C(1);
DO I = 1 TO 2 BY 1;
   A(I);
   B(I);
END;
C(2);
```

We will call such a reduced sequence consisting of do-elements, end-elements, compound-nodes, and the original nodes in S (that were not replaced by any compound node) as a schedule. The formal definition of schedule is given later in this section.

Actually, we are not interested in obtaining a schedule from the underlying graph of the array graph G. We are interested in obtaining a schedule for an array graph (or its induced graph) directly from the array graph, or at worst, from an equivalent decomposed version of the array graph. In the definition of the schedule that follows, we will use a modified form of a do-element which is given by:

```
IF p THEN DO I = 1 TO u BY c;
```

where p is a boolean expression. This boolean expression is used to restrict the range of the succeeding compound node from that of preceeding one. For example, if S contains the following sequence of nodes:

A(1); B(1); C(1); A(2); B(2); C(2); A(3); B(3); D(1); ...

Then the above sequence can be replaced by the schedule:

```
DO I = 1 TO 3 BY 1;
    A(I);
    B(I);
    IF I < 3 THEN DO J = I;
        C(J);
    END;
END;
```

In the above example, the range for J is restricted by the condition in the if-clause of the do-element preceeding C(J).

A schedule for an induced graph G_i can now be defined as follows:

(1). A(d) for any integer constant d, such that A(d,d,1) $\in$ L, is a schedule element,

(2). A(c.I+d) for a variable I and constants c and d such that A(c.l+d,c.u+d,c) $\in$ L, is an I-element.

(3). If L is a list of I-elements and schedule-elements and p is a boolean expression, then

```
IF p THEN DO I = 1 TO u BY c;
    L;
END;
```

is a schedule element.

A schedule is then a list of schedule elements such that if we unfold all loops (letting I range between 1 and u in increments of c), we obtain a topological sorting of the underlying graph of the array graph G. This implies in particular that all instances of nodes appear in the schedule, that no instance appears twice, and that any instance of an edge is represented in the ordering.

We will now define an induced graph to be <u>schedulable</u> if we can obtain a schedule for that graph.

9.2.4 The Main Problem.

Now, we are ready to pose our main problem, that is: Is it possible to decide the schedulability of a given array graph? If not, can we prove that the graph is not schedulable?

Note that, if the underlying graph corresponding to an array graph is acyclic, then that array graph is schedulable. Since we are not interested in decomposing the array graph into its equivalent underlying graph, we will define an induced graph to be <u>sequenceable</u> if the corresponding underlying graph is acyclic. We will try to obtain a necessary and sufficient condition for sequenceability of an induced graph (and not the underlying graph), and then obtain a condition for schedulability.

To expose the complexity involved in scheduling an array graph, we will now select four examples of varying complexity.

Example 1.

If the induced graph is acyclic, then the schedule can be obtained easily using conventional topological sorting techniques and by interpreting any A(l,u,c) $\in$ L as the schedule element:

```
DO I = 1 TO u BY c;
    A(I);
END;
```

Example 2.

If the induced graph contains cycles, and these cycles are due to the Ø2 edges, then some of these cycles can be sequenced using a single schedule element for each cycle. For example, for the induced graph shown in Figure 9.5, the schedule is given by:

```
A(1);
DO I = 2 TO 4 BY 1;
    A(I);
END;
```

Example 3.

If the induced graph contains cycles, and these cycles are due to the Ø2 edges, then it may not be possible to obtain the sequence for the graph using a single schedule element for each cycle. In such cases, some nodes in the cycle may have to be split. For example, consider the following array graph:

```
A(I) <- A(I+2)              for  1<= I <= 4
A(I) <- A(I-1) + A(I-5)     for  6<= I <= 10
A(5) <- C
```

The induced graph for the above array graph is shown in Figure 9.6. It is not possible to sequence the above graph without splitting the nodes connected by the first two compound edges. One possible splitting decomposes the above graph into the following:

```
A(I) <- A(I+2)                for I = 1 TO 3 BY 2
A(I) <- A(I+2)                for I = 2 TO 4 BY 2
A(6) <- A(5) + A(1)
A(I) <- A(I-1) + A(I-5)       for I = 7 TO 10
A(5) <- C
```

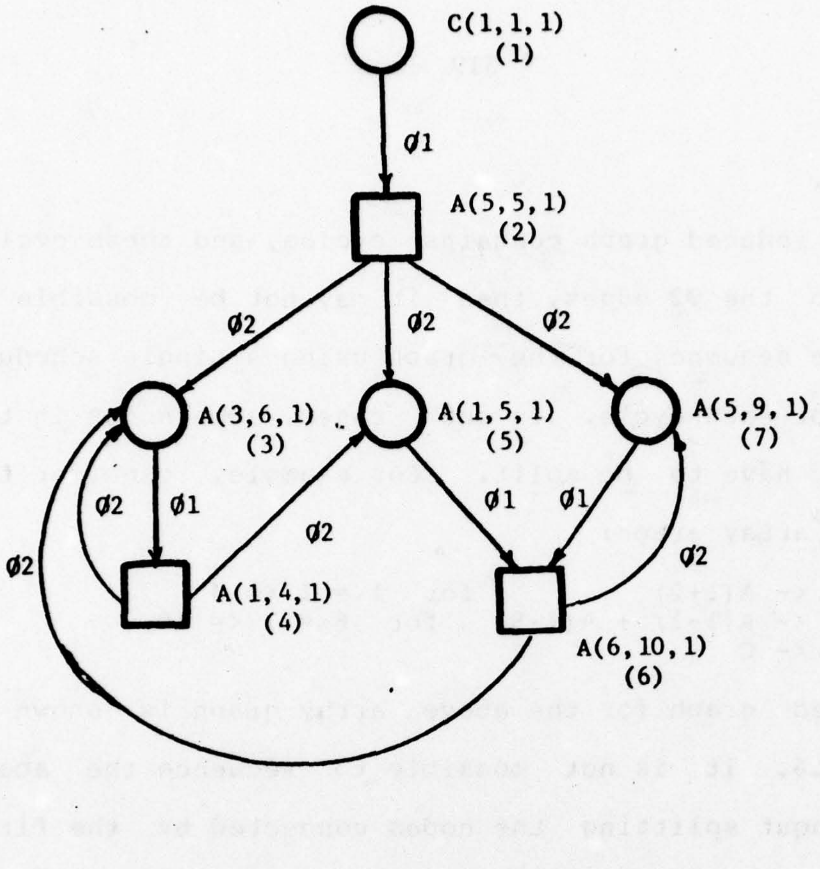


Figure 9.6. Induced Graph for the Assertions:

A(I) = A(I+2); For 1⩽ I⩽ 4
A(I) = A(I-1)+A(I-5); For 6⩽ I⩽ 10
A(5) = C;

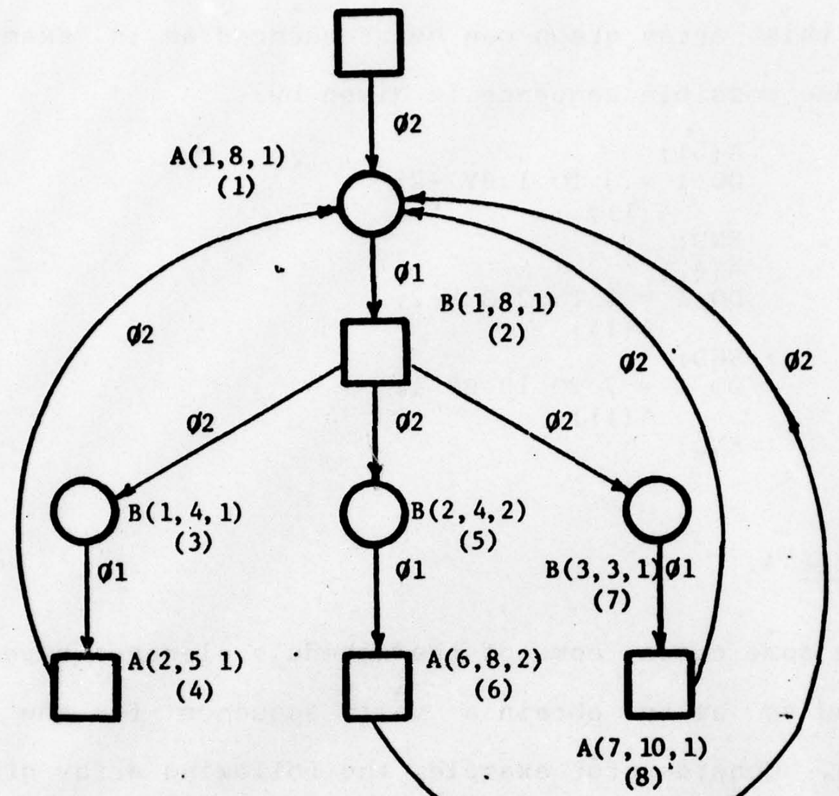


Figure 9.7. Induced Graph for the Assertions:

B(I) = A(I);	For 1 ≤ I ≤ 8
A(I+1) = B(I) + 1;	For 1 ≤ I ≤ 4
A(2.I+4) = B(2.I)+1;	For 1 ≤ I ≤ 2
A(I+6) = B(3)+1;	For 1 ≤ I ≤ 4

Now, this array graph can be sequenced as in Example 2, and one possible sequence is given by:

```
A(5);
DO I = 3 TO 1 BY -2;
    A(I);
END;
A(6);
DO I = 4 TO 2 BY -2;
    A(I);
END;
DO I = 7 TO 10 BY 1;
    A(I);
END;
```

Example 4.

In some cases, some of the schedule elements have to be merged so as to obtain a valid sequence for the array graph. Consider for example, the following array graph:

```
B(I) = A(I)                 FOR 1 <= I <= 8
A(I+1) = B(I) + 1           FOR 1 <= I <= 4
A(2.I+4) = B(2.I)+1         FOR 1 <= I <= 2
A(I+6) = B(3) +1            FOR 1 <= I <= 4
```

The corresponding Induced graph is shown in Figure 9.7. One possible schedule for this graph is given by:

```
DO I1 = 1 TO 8 BY 1;
     B(I1);
     IF I1 <= 4 THEN DO I2 = I1;
          A(I2+1) = B(I2) + 1;
     END;
     IF (I1 >= 2 & I1 <= 4 & MOD(I1,2) = 0 )
            THEN DO I3 = I1;
          A(I3+4) = B(I3) + 1;
     END;
     IF I1 = 3 THEN DO I4 = I1 + 4 TO I1 + 7;
          A(I4) = B(3) + 1;
     END;
END;
```

Note that the boolean expressions associated with the inner do-loops restrict the associated iteration to a subrange of the range of subscript associated with the outer most loop.

From the above examples, we can see that the conventional topological sorting algorithm cannot be applied directly to every induced graph. Some graphs have to be decomposed, and in such cases, we would like to minimize the amount of decomposition, because, the more compact the graph, the more efficient will the schedule be. We will not try to present an "efficient" algorithm for the decomposition of "decomposable" array graphs, but will derive some criterions for decomposition.

9.2.5 Map Vector and Threshold.

If two nodes i and j are connected by a Ø2 edge, then we will associate a Map vector M_{ij} and a Threshold T_{ij}

with that edge which are defined as follows:

$M_{ij}(k) = 1 + q$ if there exists an integer q such that $l_i + k.c_i = l_j + q.c_j$ for $0 <= k <= \lfloor (u_i - l_i)/c_i \rfloor$ and $0 <= q <= \lfloor (u_j - l_j)/c_j \rfloor$

= 0, otherwise.

and

$T_{ij} = MIN(M_{ij}(k) - k)$ for every k for which $M_{ij}(k) > 0$.

As an example, for the induced graph in Figure 9.5,

$M_{23} = (1)$,

$T_{23} = 0$,

$M_{43} = (2,3,0)$,

and $T_{43} = MIN(2-1,3-2) = 1$.

The Threshold and the Map vector are useful in the analysis of cycles that contain Ø2 edges. In such cycles,

---

From now on, we will represent the range of a numbered node by using appropriate subscripts. For example, i th node will be represented by: $A(l_i, u_i, c_i)$

some of the underlying elements of a L node must precede some others, either by direct or indirect implication. The precedence can be between elements of lower subscripts and elements of higher subscripts, or vice versa, leading to an ascending or descending loop respectively. Thus, in the example of Figure 9.5, A(2) must precede A(3), and A(3) must precede A(4). If such a precedence always holds, then we can say that the loop is "consistent". The Threshold measures the distance between the closest elements which should be related by precedence.

9.2.6 Map Vector between two arbitrary nodes.

If two nodes i and j are connected by a sequence of Ø1 and/or Ø2 edges, then we will associate a map-vecotr M_{ij} between the two nodes which is defined as follows:

(1). If k is a successor of node i and k is in the Path(i,j) then, if i and k are connected by a Ø2 edge, then the map vector M_{ik} is defined as in the previous section. Otherwise,

M_{ik} = (1,2,3, ... n),

where, $n = 1 + \lfloor (u_i - l_i)/c_i \rfloor$

(2). If k and l are two nodes in the path(i,j) such that path(i,k) < path(i,l)[c] <= path(i,j), then

M_{il} is defined as follows:

If k is connected to l by a Ø1 edge, then

$M_{il} = M_{ik}$;

otherwise,

$M_{il}(q) = M_{kl}(M_{ik}(q))$, if $M_{ik}(q) > 0$

= 0, otherwise.

for all q in the range: $1 <= q <= n$.

If there are more than one paths between the two nodes, then the map-vector is superscripted with the sequence of nodes that represents the appropriate path.

As an example, for the induced graph shown in Figure 9.7,

$M_{23} = (1,2,3,4,0,0,0,0)$

Path(i,j) denotes a path from vertex i to vertex j and is defined as the sequence of edges:

(i,v1) (v1,v2) ... (vn,j)

or by the sequence of nodes: i v1 v2 ... vn j

where, v1, v2, ..., vn are the nodes in the path from vertex i to vertex j. Also, if one path is contained in another, then one of the relational operators is used to indicate such relation. Therefore, in the above example, path(i,v2) < path(i,j).

$M_{25} = (0,1,0,2,0,0,0,0)$

$M_{41} = (2,3,4,5)$

$M_{62} = (6,8)$

$M^{2341}_{21} = (2,3,4,5,0,0,0,0)$

$M^{2561}_{21} = (0,6,0,8,0,0,0,0)$

...

and so on.

M_{ij} determines the precedence between different elements of the two nodes. It also indicates that if the q th element of M_{ij} is non zero, then there exists a path in the underlying graph from the node represented by the q th element of node i, that is,

$A(1 + c_i.(q-1))_i$

to the node represented by the $M_{ij}(q)$ th element of node j, that is,

$B(1_j + c_j.(M_{ij}(q) - 1))$

9.2.7 Self Map Vector.

For a node i in a cycle of an induced graph, we define the self map vector Q_i as equal to M_{ii}^{x}, where x is the sequence of nodes (path(i,i)) in the cycle. As in the case of map vector, if there are more than one cycles through the node i, then we will distinguish different self map vectors by using appropriate superscripts.

As an example, for the induced graph shown in Figure 9.7, we have:

$$Q_1^{12341} = M_{21}^{2341} = (2,3,4,5,0,0,0,0)$$

$$Q_1^{12561} = M_{21}^{2561} = (0,0,0,8,0,0,0,0)$$

$$Q_3 = (2,3,4,0)$$

As in the case of the map vector, elements of the loop map vector Q_i determines the precedence between different elements of node i. In fact, if $Q_i(q)$ is non-zero, then it means that $Q_i(q)$ th element of node i depends on the q th element of node i. Therefore, for the cycle to be consistent, the following relation must be true for every node i in the cycle:

$Q_i(q) > q$ for every integer q for which $Q_i(q) > 0$

In the induced graph of Figure 9.7, all the three cycles satisfy the above condition, and therefore are consistent.

9.2.8 Representation of Map and Self Map Vectors.

If V represents a set of elements that represents a map vector or a self map vector, then its non-zero elements are either monotonically increasing or decreasing. Therefore, we can represent such a vector using two triplets as given below:

$$V = [l_x, u_x, c_x]\ [l_y, u_y, c_y]$$

where-

l_x is the smallest integer for which $V(l_x) > 0$,

u_x is the largest integer for which $V(u_x) > 0$,

$c_x = \lfloor (u_x - l_x)/n \rfloor$ if n = the number of non-zero elements in V is greater than 1;

$= 1$, otherwise,

$l_y = V(l_x)$,

$u_y = V(l_x)$,

and finally,

$c_y = \lfloor (u_y - 1)/n_y \rfloor$ if $n > 1$,

$= 1$, otherwise.

For example, the vector (4,0,2,0) will be represented by: [1,3,2][4,2,-2]; whereas, the vector (0,0,1,2,3,4,0) will be represented by: [3,6,1][1,4,1].

As another example, consider the induced graph shown in Figure 9.7.1. The map vectors associated with the Ø2 edges and the self map vector associated with node (1) of the graph in this figure is shown in Figure 9.7.2. These vectors in the 6-tuple notation (described above) are given by:

M_{23} = [1,4,3][4,2,-2]

M_{41} = [1,4,1][2,5,1]

and Q_1 = [1,4,3][5,3,-2]

With the above definitions, we can now start out analysis of induced graph.

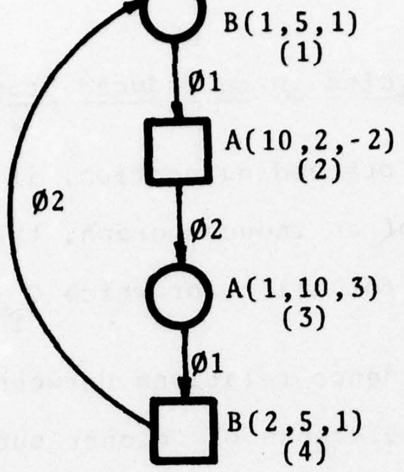


Figure 9.7.1. A Simple Induced Graph.

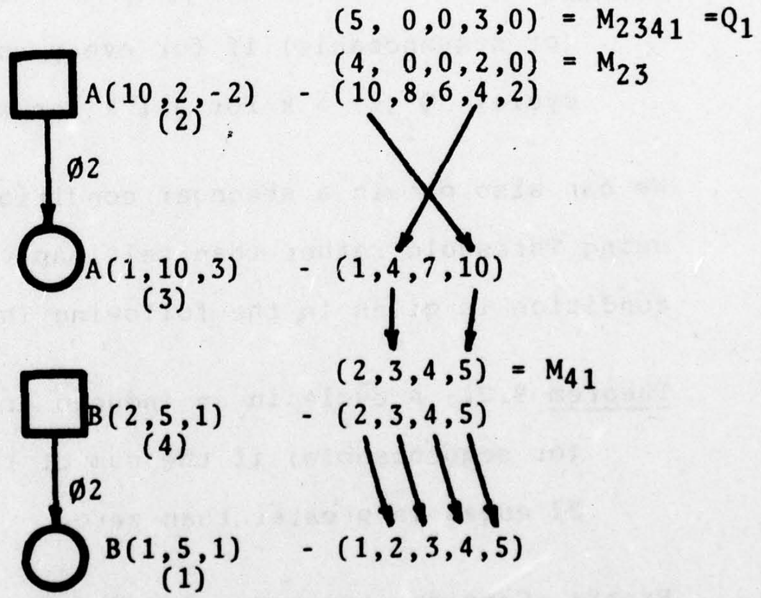


Figure 9.7.2. Map and Self Map vectors for the Graph in Figure 9.7.1.

9.2.9 Consistency of Cycles in an Induced Graph.

As discussed in the preceeding section, if for every node $i \in R$ in a cycle of an induced graph, the relation $Q_i(k) > k$ is satisfied for all k for which $Q_i(k) > 0$, then there exists precedence relations between elements of lower subscripts and elements of higher subscripts, or vice versa. Therefore, such a cycle can be sequenced; and so, the following theorem is proven:

__Theorem__ 9.1. A cycle in an induced graph is consistent (or sequenceable) if for every node $i \in R$ in the cycle, $Q_i(k) > k$ for all k for which $Q_i(k) > 0$.

We can also obtain a stronger condition for consistency using Threshold rather than self map vector. This condition is given in the following Theorem:

__Theorem__ 9.2. A cycle in an induced graph is consistent (or sequenceable) if the sum of thresholds on all its Ø2 edges is greater than zero.

__Proof__: Consider two nodes i and j in a cycle that are connected by a Ø2 edge. Now,

$$M_{ij}(k) = k + M_{ij}(k) - k \qquad \text{for } 1 <= k <= \lfloor (u_i - 1_i)/c_i \rfloor$$

And, by using the definition of Threshold,

$$M_{ij}(k) >= k + T_{ij}$$

$$M_{ij}(k) - k >= T_{ij}$$

We can extend the same reasoning for two nodes i and j connected by a sequence of Ø1 and Ø2 edges, and will get:

$$M_{ij}(k) - k >= T$$

where T is the sum of thresholds on all Ø2 edges in the path(i,j). Therefore, we get:

$$M_{ii}(k) - k >= T$$

or

$$Q_i(k) - k >= T$$

Therefore, if T is greater than zero, then $Q_i(k) - k > 0$, and using theorem 9.1, the cycle is consistent.

QED.

Note that, the conditions in theorems 9.1 and 9.2 are equivalent if the cycle contains only one Ø2 edge. However, the condition for consistency in theorem 9.1 is weaker than the condition in theorem 9.2 for a general case. In other words, there exists cycles which are consistent with respect to theorem 9.1, but the condition in theorem 9.2 does not hold. For example, for the following array graph,

```
        A(I+9) <- B(I)              1 <= I <= 12
        B(I-3) <- A(I)              8 <= I <= 20 BY 2
```

the induced graph is shown in Figure 9.8. For this graph,

$M_{23} = (2,0,3,0,4,0,5,0,6,0,7,0)$

$T_{23} = \mathrm{MIN}(2-1,7-11) = -4$

$M_{41} = (5,7,9,11,0,0,0)$

$T_{41} = \mathrm{MIN}(5-1,11-4) = 4$

$Q_{1} = (7,0,9,0,11,0,0,0,0,0,0,0)$

Therefore, $Q_1(k) > 0$ for all k for which $Q_1(k) > 0$;

and so, the cycle is consistent from theorem 9.1. However, the consistency condition in theorem 9.2 is not satisfied, because, the sum of thresholds associated with all the Ø2 edges in the cycle is zero.

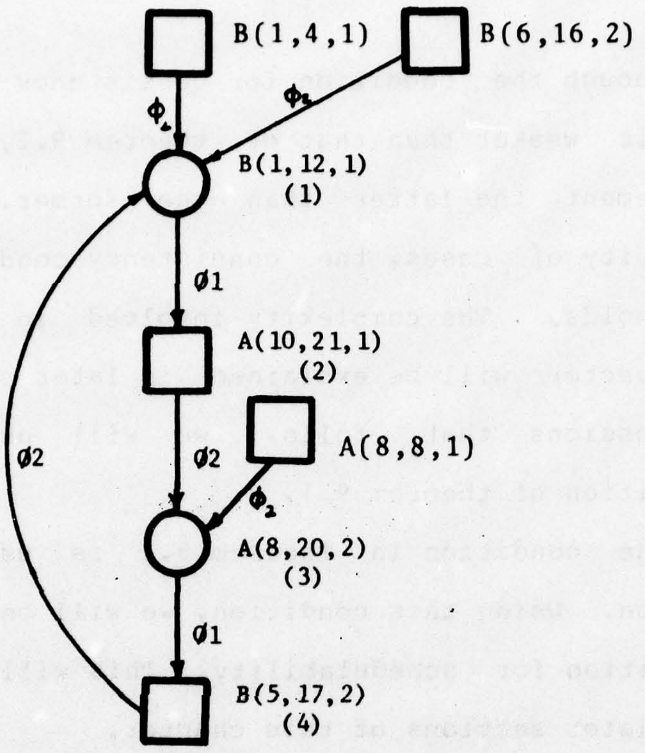


Figure 9.8. Induced Graph for the Assertions:

A(I+9) = B(I);	For 1 ≤ I ≤ 12
B(I-3) = A(I);	For I = 8 TO 20 BY 2

Though the condition for consistency given in theorem 9.1 is weaker than that of theorem 9.2, it is easier to implement the latter than the former. Also, for the majority of cases, the consistency condition of theorem 9.2 holds. The complexity involved in computing a self map vector will be explained in later sections. In the discussions that follow, we will use the (weaker) condition of theorem 9.1.

The condition in Theorem 9.2 is useful for another reason. Using this condition, we will be able to derive a condition for schedulability. This will be described in the later sections of this chapter.

Note that the consistency condition (of Theorem 9.1) is not a necessary condition for the sequenceability of a cycle in an induced graph. As an example, consider the induced graph shown in Figure 9.6. For this graph,

$$M_{43} = (0,0,1,2)$$

$$Q_3^{343} = (0,0,1,2)$$

Therefore the cycle is inconsistent according to theorem 9.1. But the cycle (or the graph) is sequenceable as was shown in section 9.2.4. Let us now try to analyze the array graphs, which consist of inconsistent cycles, and obtain a necessary and sufficient condition for sequenceability after decomposing some nodes in those graphs (if necessary).

9.2.10 Decomposition of nodes in the Array Graphs.

If a cycle in an induced graph is inconsistent, then one way to test the sequenceability of the cycle is to test the sequenceability of the underlying graph corresponding to the cycle. Such a test is laborious, and is not interesting. However, it is possible to decompose the cycle and test its sequenceability using a more efficient algorithm. In the following, two criterions used to decompose inconsistent cycles in an induced graph are described.

9.2.10.1 Decomposition Criterion(1).

If a node $i \in R$ is contained in a cycle and D_i denotes the set of elements represented by the node, that is,

$$D_i = (l_i, l_i + c_i, l_i + 2.c_i, \ldots, l_i + n.c_i)$$

$$\text{where } n = \lfloor (u_i - l_i)/c_i \rfloor$$

and if S_i denotes the set

$$S_i = (D_i \cap (D_{p1} \cup D_{p2} \cup \ldots \cup D_{pn}))$$

where p1, p2, ..., pn are the Ø2 predecessors of node i, that are not contained in any of the cycles through the node i, then the set S_i must contain the leading elements of the set D_i; otherwise, the node i (and all the related nodes) must be decomposed till the above condition is met.

Loosely speaking, the above criterion restricts every source variable node of an assertion to contain all the "computable" elements (elements that are obtained from outside the cycle, or the "external stimuli" to the cycle) to be the elements with lowest or highest subscripts according as the increment associated with the subscript is positive or negative respectively.

If the set S_i represents the trailing elements of the set D_i (rather than the leading elements), then no decomposition is necessary, but the increment associated with the node i (and all the related nodes) must be reversed. This reversal rotates the elements of the set D_i, and therefore, makes the set S_i to represent the leading elements of set D_i.

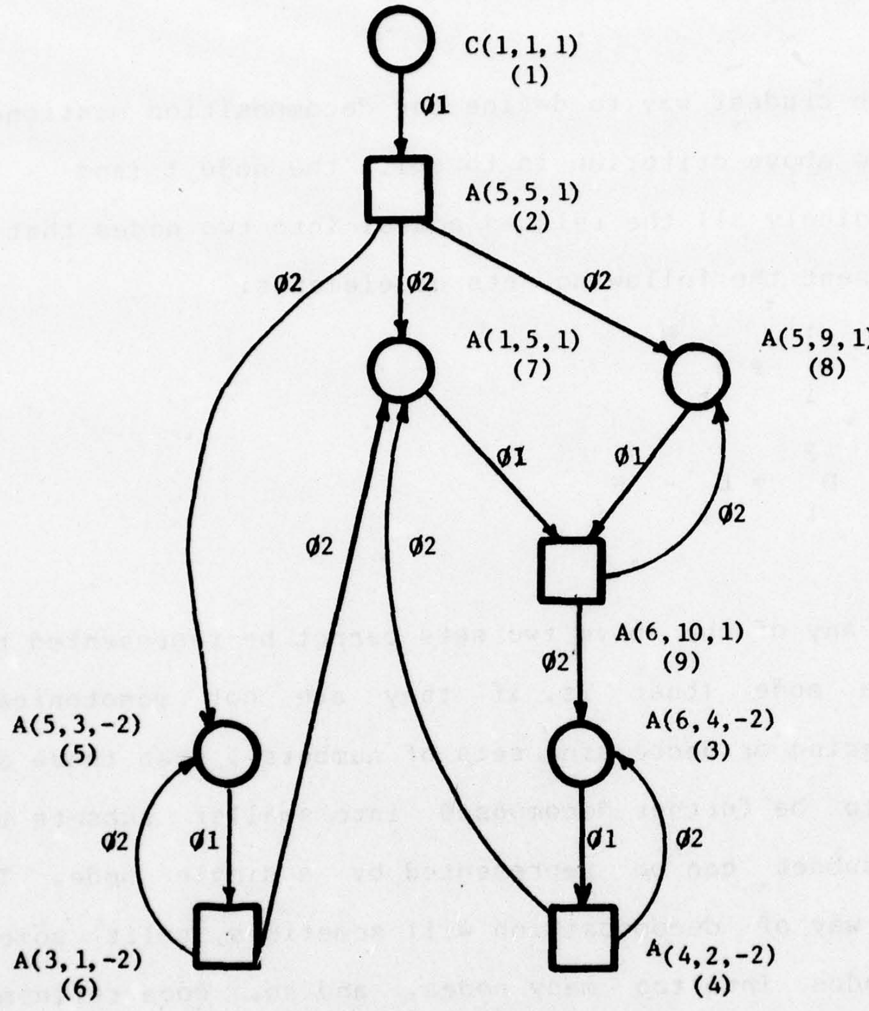


Figure 9.9. A decomposed Graph of Figure 9.6.

The crudest way to define the decomposition mentioned in the above criterion is to split the node i (and accordingly all the related nodes) into two nodes that represent the following sets of elements:

$$D_i^1 = S_i$$

and $$D_i^2 = D_i - S_i$$

If any of the above two sets cannot be represented by a single node (that is, if they are not monotonically increasing or decreasing sets of numbers), then those sets have to be further decomposed into smaller subsets till each subset can be represented by a single node. This crude way of decomposition will sometimes, split some of the nodes into too many nodes, and so, some refinement must be used. Such refinements will be described later.

We will now apply the above decomposition criterion to the induced graph shown in Figure 9.6. The node (3) in the graph can be decomposed, and one possible decomposition is given by:

A(6,4,-2)

and A(5,3,-2)

The decomposed graph is shown in Figure 9.9. All the cycles in this graph are consistent w.r.t. the theorem 9.1.

The decomposition criterion described above is not sufficient to obtain a necessary and sufficient condition for sequenceability. For example, consider the induced graph shown in Figure 9.10. This graph is indecomposable using criterion (1). Also, for this graph,

$$M_{34} = (1,0,2,0)$$

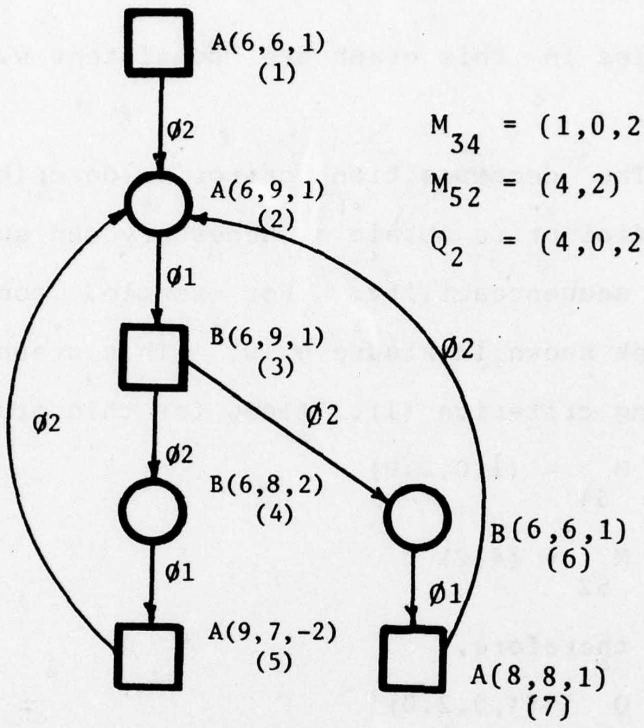
$$M_{52} = (4,2)$$

and therefore,

$$Q_{2} = (4,0,2,0)$$

and so, the cycle is inconsistent. But the induced graph is schedulable, and one possible schedule for the graph is given by:

```
A(6);
DO I = 6 TO 8 BY 2;
     B(I);
     A(15-I);
     IF I = 6 THEN DO;
        A(8);
     END;
END;
DO I = 7 TO 9 BY 2;
     B(I);
END;
```



$$M_{34} = (1,0,2,0)$$

$$M_{52} = (4,2)$$

$$Q_2 = (4,0,2,0)$$

Figure 9.10. Induced Graph for the Assertions:

```
A(6) = 10;
B(I) = A(I);            For  6 ≤ I ≤ 9
A(8) = B(6);
A(15-2.I) = B(2.I);   For  3 ≤ I ≤ 4
```

In the above schedule, the nodes (2) and (3) are split into two sets of nodes A(6,8,2), A(7,9,2) and B(6,8,2), A(7,9,2) respectively.

Theorem 9.3. If the self map vector associated with a node i $\in$ R in a cycle of an induced graph is given by:

$$Q_i = [l_x, u_x, c_x][l_y, u_y, c_y]$$

where both l_x and c_x are 1; and if the cycle is indecomposable w.r.t. the criterion (1), then the cycle is sequenceable iff for every node i $\in$ R in the cycle, $Q_i(q) > q$ for all q for which $Q_i(q) > 0$.

Proof:

If both l_x and c_x are 1 then it means that $Q_i(p) > 0$, for all p in the range $0 < p <= u_x$. Also, if there exists an integer q for which $q >= Q_i(q) > 0$, then either, (1) $l_y = 1$,

or, (2) $c_y < 0$.

In the first case, the cycle is obviously inconsistent and not sequenceable, because, the first element of the node is its own predecessor, so the underlying graph itself is not acyclic.

In the second case, the precedence is between the elements of higher index to that of lower index (or vice versa if increment is < 0), even though the leading elements of the node were computable. Even in this case, the cycle cannot be decomposed to a set of consistent cycles. In other words, at least one element of the node is its own predecessor, directly or indirectly, implying that the underlying graph itself is not acyclic.

QED.

From the above theorem, we can obtain another decomposition criterion which is described in the following.

9.2.10.2 <u>Decomposition Criterion</u>(2).

If for a node $i \in R$ in a cycle of an induced graph, $Q_i = [l_x, u_x, c_x][l_y, u_y, c_y]$ and $l_x > 1$ or $c_x > 1$, then decompose the node i (and all the related nodes) into the nodes represented by the following set of elements:

$$D_i^1 = A(1 + k.c) \mid 0 <= k < l_x - 1$$

$$D_i^2 = A(1 + k.c.c_x) \mid$$

$$l_x - 1 <= k <= \lfloor (u_x - (1+c.(l_x - 1)))/c_x . c \rfloor$$

and $D_i^3 = D_i - (D_i^2 + D_i^1)$

where-

the node i is represented by A(1,u,c).

As an example, consider the directed graph shown in Figure 9.10. For this graph,

$$Q_2 = (4,0,2,0) = [1,3,2][4,2,-2]$$

Therefore, using the criterion (2), the node 2 can be decomposed into nodes that represents the following sets of elements:

$$D_i^1 = ()$$

$$D_i^2 = (6,6+2) = (6,8)$$

and $D_i^3 = (6,7,8,9) - (6,8) = (7,9)$

The decomposed graph is shown in Figure 9.11.

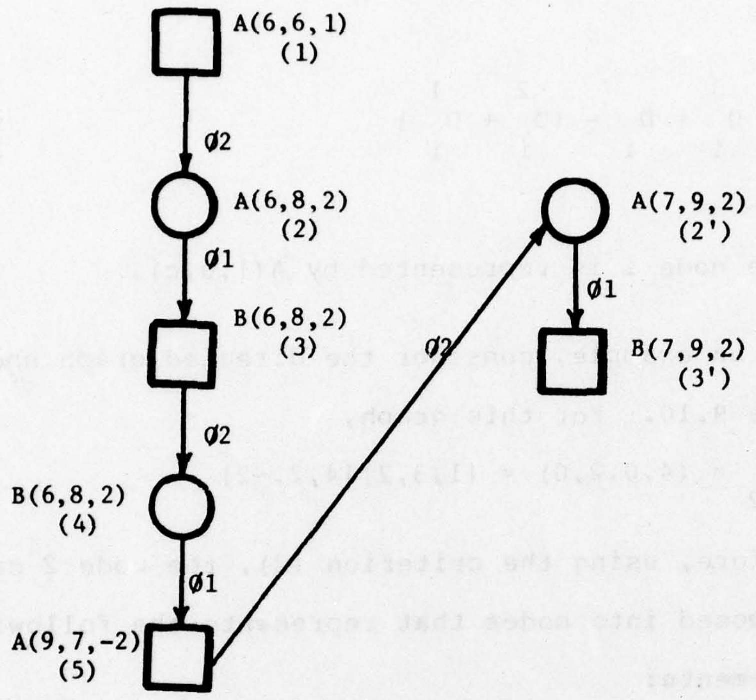


Figure 9.11. A Decomposed Graph of Figure 9.10.

We will now say that an induced graph is <u>indecomposable</u> if the decomposition criterions (1) and (2) do not decompose the graph. Therefore, Theorem 9.3 can be rephrased as follows:

<u>Theorem</u> 9.4.

If an induced graph is indecomposable, then it is sequenceable iff for every node $i \in R$ of a cycle in the graph, $Q_i(q) > q$ for all q for which $Q_i(q) > 0$.

9.2.11 The ASPLIT1 Procedure.

The ASPLIT1 procedure is used to decompose a node of an induced graph using the decomposition criterion (1) described in section 9.2.10. The algorithm A9.1 describes the functions of the routine ASPLIT1. The steps 1 thru 9 in the algorithm, obtain the first (X1) and the last (X2) "computable" elements of the node, and the first (Y1) and the last (Y2) "non-computable" elements of the node.

If X1 < Y1 and X2 < Y1, then all the computable elements are the leading elements of the node; and so, the

---

"computable" elements of a node $i \in R$, are those elements that are contained in those Ø2 predecessors of i which are not contained in any of the cycles through i; and "non-computable" elements are those elements that are not computable.

decomposition criterion (1) cannot be applied (step 10).

If the computable elements of a node are the trailing elements of that node, then the increment is negated (step 11). The negation is also performed if the "distance" between the first element and X1 is greater than the distance between the last element and X2. An example of the latter case is the node (3) in the induced graph of Figure 9.6. For this node:

X1 = 3 = X2,

Y1 = 1, and

Y2 = 4.

In this case, the increments associated with the nodes (3) and (4) are negated, so that their representations are changed to A(6,3,-1) and A(4,1,-1) respectively.

If it is possible to decompose a node, then one of the three techniques for decomposition described in steps (12), (16) and (19) is used.

<u>ALGORITHM</u> A9.1: ASPLIT1(I).

```
Function: Decomposes the node I of an induced graph using
          the decomposition criterion (1) described in
          section 9.6.1.
Step1:  Set X1, X2, Y1, Y2 = 0.
Step2:  For each Ø2 predecessor, K, of node I, perform
```

steps (3) thru (9).

Step3: If K is contained in a cycle through I, then goto step 7.

Step4: Let $M_{ki} = [l_x, u_x, c_x][l_x, u_x, c_x]$. If X1 = 0 then set X1= $MIN(l_y, u_y)$; otherwise, set X1 = $MIN(X1, l_y, u_y)$.

Step5: If X2 = 0 then set X2 = $MAX(l_y, u_y)$; otherwise, set X2 = $MAX(X2, l_y, u_y)$.

Step6: Goto step 9.

Step7: If Y1 = 0 then set Y1 = $MIN(l_y, u_y)$; otherwise, set Y1 = $MIN(Y1, l_y, u_y)$.

Step8: If Y2 = 0 then set Y2 = $MAX(l_y, u_y)$; otherwise, set Y2 = $MAX(Y2, l_y, u_y)$.

Step9: Continue.

Step10: If X1 < Y1 and X2 < Y1 then no decomposition, so, return.

Step11: If (X2 > Y2 & X1 > Y2) | (Y2 > X2 & Y2-X2 < X1-Y1) Then negate the increment associated with this node and all the related nodes. At end, return.

Step12: If X1 = 1 then split the node I into two nodes that represent the following sets of elements:

$$D_i^1 = A(l_i), A(l_i + c_i), \ldots, A(l_i + (Y1-2).c_i)$$

and

$$D_i^2 = A(l_i + (Y1-1).c_i), A(l_i + Y1.c_i), \ldots,$$

$$A(l_i + k.c_i)$$

$$\text{where } k = \lfloor (u_i - l_i)/c_i \rfloor.$$

At end, return.

Step13: For each cycle through I, perform steps 14 thru 18.

Step14: Compute Q_i for the cycle; and if $Q_i(X1) > 0$

and $Q_i(X1) > X1 + 1$ then do steps 15 thru 17.

Step15: Let $J = Q_i(X1)-X1$.

Step16: Decompose the node I into two nodes representing the following sets of elements:

$$D_i^1 = A(l_i + (X1-1).c_i), A(l_i + (X1+J-1).c_i), \ldots,$$

$$A(l_i + (X1+k.J-1).c_i)$$

$$\text{where } k = \lfloor ((u_i - l_i)/c_i + 1 - X1)/J \rfloor$$

and

$D_i^2 = D_i - D_i^1$.

Decompose D_i^2 if $X1 \neq J$ or $X1 = J$ & $J > 2$.

At end, return.

Step17: Continue.

Step18: Continue.

Step19: Decompose the node I into two nodes represented by the following sets of elements:

$D_i^1 = A(1_i), A(1_i + c_i), \ldots, A(1_i + (X1-2).c_i)$

and

$D_i^2 = D_i - D_i^1$.

Step20: Return.

9.2.12 <u>The</u> <u>ASPLIT2</u> <u>Procedure</u>.

The procedure ASPLIT2 is used to decompose a node of an induced graph using the decomposition criterion (2) described in section 9.6.1. The algorithm A9.2 describes the functions of the routine ASPLIT2.

ALGORITHM: A9.2: ASPLIT2(I).

Function: Decomposes the node I in an induced graph using the decomposition criterion (2) described in section 9.6.1.

Calls:

Called by:

Step1: For each Ø2 predecessor, K, of node I, perform Steps (2) thru (6).

Step2: Let $M_{ki} = [l_x, u_x, c_x][l_y, u_y, c_y]$

Step3: If K is not contained in any cycle through I, then goto step 6.

Step4: If $l_x > 1$ or $c_x > 1$ then do step 5.

Step5: Decompose the node I into three nodes the elements of which are represented by:

$$D_i^1 = A(l + k.c) \mid 0 <= k < l_x - 1$$

$$D_i^2 = A(l + k.c.c_x) \mid$$

$$l_x - 1 <= k <= \lfloor (u_x - (l + c.(l_x - 1))/c_x .c \rfloor$$

and $D_i^3 = D_i - (D_i^2 + D_i^1)$

where-

the node i is represented by A(l,u,c).

```
        At end, return.
Step6:  Continue.
Step7:  Return.
```

9.2.13 Comments on the Decomposition of Array Graphs.

The decomposition algorithms described in earlier sections may not be too efficient for general array graphs. Some additional refinements can be applied to both of them. One refinement to the decomposition criterion (1) is given below.

Consider two cycles thru a node $i \in R$ in an array graph. If one of the cycles contains only one Ø2 edge, and the other more than one, then the decomposition criterion(1) can be applied to the sub-graph of the array graph which consists of only the former cycle. For example, consider the directed graph shown in Figure 9.12. In this figure, the cycle 1-2 is inconsistent, as it does not satisfy the condition in theorem 9.2. If we apply the above refinement for the decomposition criterion(1), the graph in Figure 9.12 will be changed to the graph shown in Figure 9.13. The cycle 1-2 in Figure 9.13 is now consistent.

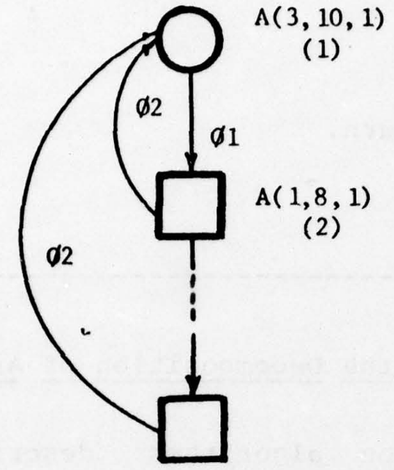


Figure 9.12.

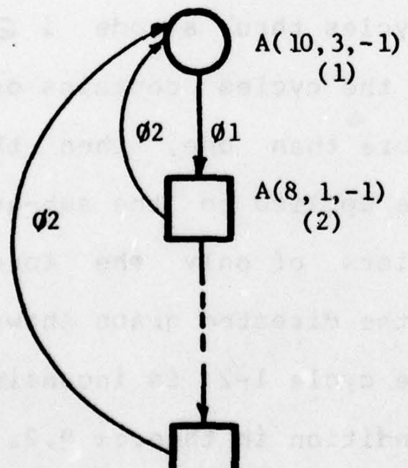


Figure 9.13.

AD-A063 900 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN--ETC(U)
SEP 78 S K SHASTRY N00014-76-C-0416
UNCLASSIFIED 78-01 NL

5 OF 6
AD
A063900

9.2.14 Sequenceability and Schedulability.

So far, we have discussed the condition for sequenceability. Now we will try to obtain an algorithm for schedulng a sequenceable graph. We will call an induced graph schedulable if we can obtain a schedule for that graph. It can be easily seen that not al sequenceable graphs are schedulable. However, we will obtain a sufficient condition for schedulability, and try to find a decomposition criterion to decompose a sequenceable graph into a schedulable graph.

To illustrate that not all sequenceable graphs are schedulable, consider the following array graph:

```
A1:  A(1) = 0;
A2:  A(2.I) = A(I) + 1              FOR 1 <= I <= 50
A3:  A(2.I-1) = A(2.I) + 2          FOR 31 <= I <= 50
A4:  A(2.I+1) = A(2.I+5) + 4        FOR 1 <= I <= 29
```

The corresponding induced graph is shown in Figure 9.14. If we apply the modified decomposition criterion(1) (see section 9.2.12), the induced graph shown in Figure 9.14 will be changed to the graph shown in Figure 9.15. For this graph, the first decomposition criterion can no longer be applied. However, second decomposition can be applied because:

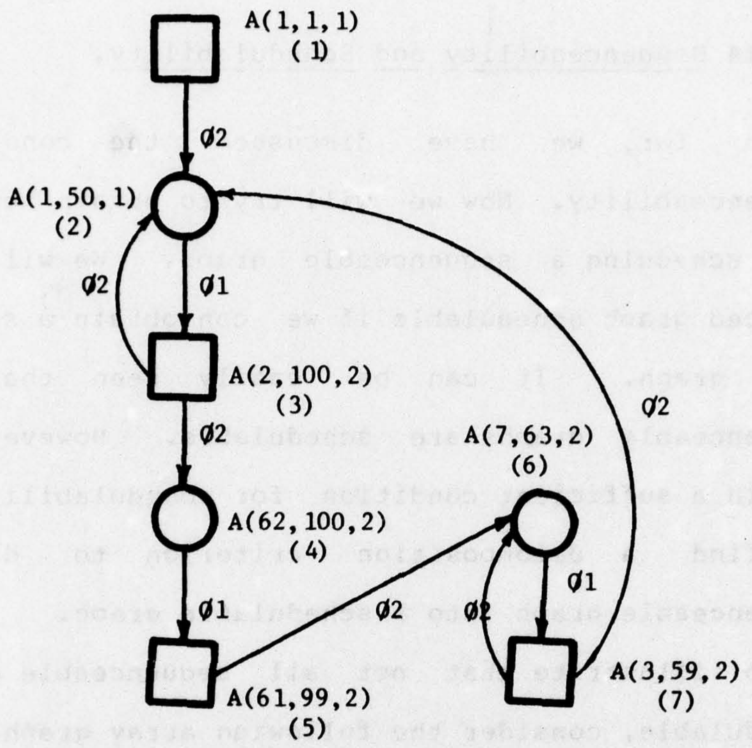


Figure 9.14. Induced Graph for the Assertions:

A(1) = 0;	
A(2.I) = A(I) + 1;	For $1 \leq I \leq 50$
A(2.I-1) = A(2.I)+2;	For $31 \leq I \leq 50$
A(2.I+1) = A(2.I+5)+4;	For $1 \leq I \leq 29$

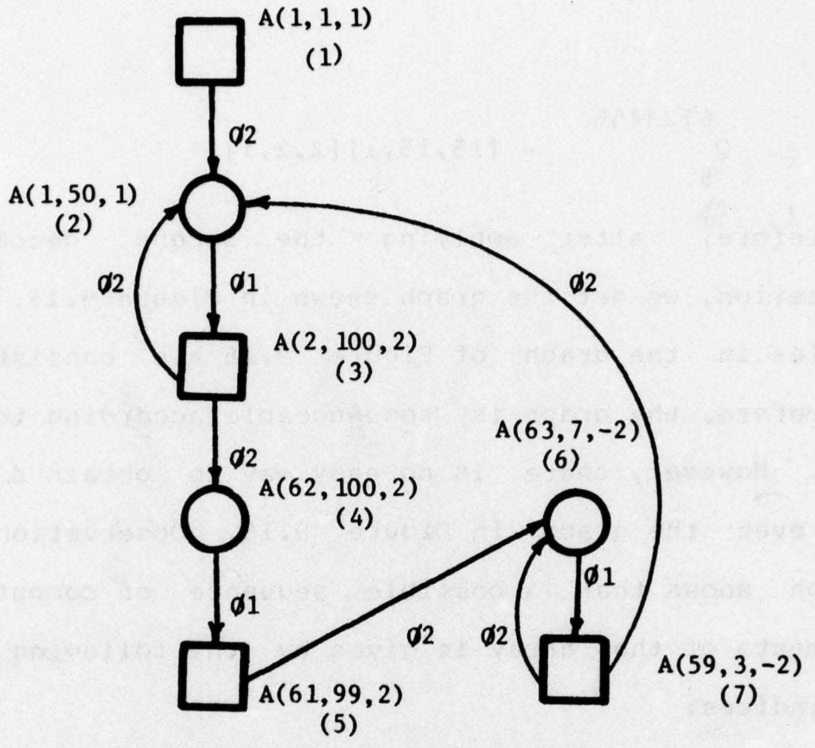


Figure 9.15. A Transformation of Graph in Figure 9.14.

$$Q^{6723456}_{6} = [15,15,1][2,2,1]$$

Therefore, after applying the second decomposition criterion, we get the graph shown in Figure 9.16. All the cycles in the graph of Figure 9.16 are consistent, and therefore, the graph is sequenceable according to theorem 9.3. However, there is no easy way to obtain a schedule for even the graph in Figure 9.16. Observation of the graph shows that a possible sequence of computing the elements of the array is given by the following sequence of indices:

1,2,4,8,16,32,64,	using A2,
63	using A3,
59,55,51,...,31,...7,3,	using A4,
6,14,22,...,62,...,94,	using A2,
61	using A3,
57,53,49, ... 5	using A4,

...

and so on.

Therefore, the graph has to be decomposed further so as to schedule it. A decomposition criterion to decompose sequenceable graphs into schedulable graphs is derived in the following sections.

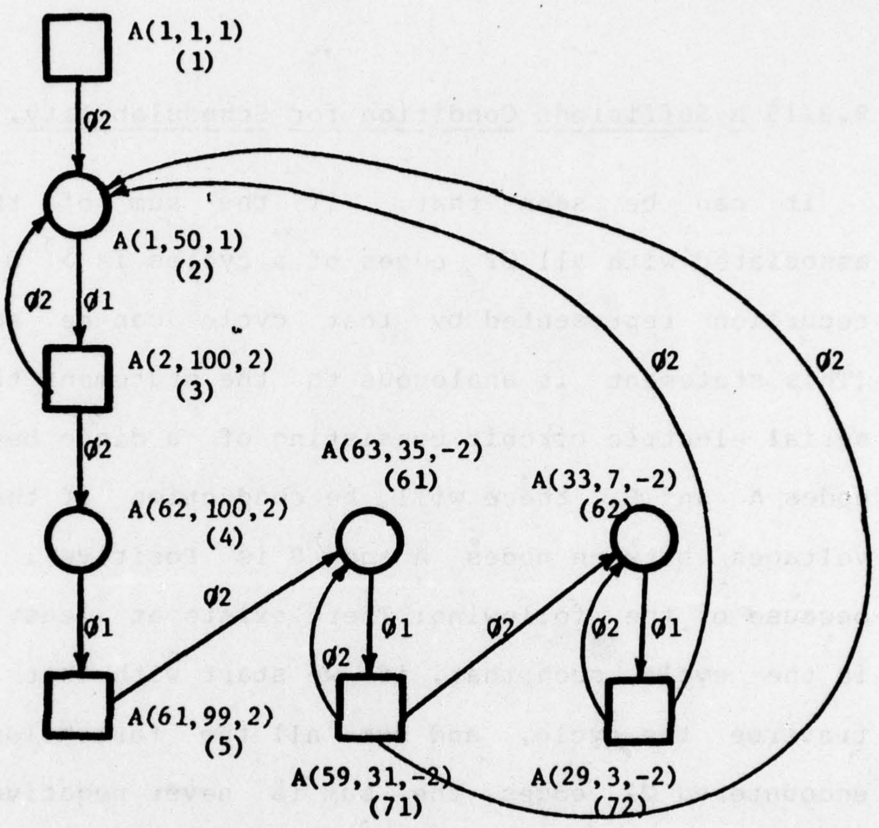


Figure 9.16. A Decomposition of Graph in Figure 9.15.

9.2.15 A Sufficient Condition for Schedulability.

It can be seen that, if the sum of thresholds associated with all Ø2 edges of a cycles is > 0 then the recursion represented by that cycle can be scheduled. (This statement is analogous to the statement that in a serial electric circuit consisting of a diode between two nodes A and B, there will be conduction if the sum of voltages between nodes A and B is Positive). This is because of the following: There exists at least one node in the cycle, such that, if we start with that node and traverse the cycle, and sum all the thresholds on the encountered Ø2 edges, the sum is never negative. This means that, if we start to generate the schedule starting with that node, we can obtain the schedule for that cycle. Therefore, we get the following condition for schedulability:

> An induced graph is schedulable, if the sum of thresholds associated with all the Ø2 edges is positive for every cycle in the graph.

Algorithm A9.3 describes the steps involved in generating a schedule for a sequenceable induced graph. This algorithm will be illustrated by going thru the sequence of steps while generating the schedule for the graph in Figure 9.7. For this example, START = 1 and END = 1; and in step 1, 3 cycles are obtained. These are: (1,2,3,4), (1,2,5,6) and (1,2,7,8). The range (ls,us,cs) is (1,8,1) in step 3. During the analysis of the cycle (1,2,3,4), in steps (5) thru (13), the following schedule elements are generated:

```
        DO Is = 1 TO 8;
        B(Is);
        END;
        IF Is <= 4 THEN DO I3 = Is;
        A(I3+1);
        END;
```

During the analysis of the cycle (1,2,5,6), the following schedule elements are generated:

```
    IF Is >= 2 & Is <= 4 & MOD(Is,2) = 0 THEN DO I5 = Is;
    A(I5+4);
    END;
```

And, during the analysis of the cycle (1,2,7,8), the following schedule elements are generated:

```
        IF Is >= 3 & Is <= 6 THEN DO I7 = Is;
        A(I7+4);
        END;
```

And, finally, an end-element is generated in step 14.

<u>Algorithm</u> A9.3. SCHEDUL (START,END);

Function: Generates the schedule for a set of nodes that are strongly connected. START and END are the indices of the order vector, ORDERV, representing the beginning and end of the set of strongly connected nodes.

Step1: Find a list of all cycles in the set of nodes represented by ORDERV(START) thru ORDERV(END). If there are no cycles, then return.

Step2: Let SEQUENCED be an array of size #DICT, and initialize all the elements in it to zeros.

Step3: Let the range(ls,us,cs) cover all the ranges of the subscript Is associated with every cycle obtained in step 1. Also, generate the following do-element for the START node:

DO Is = ls TO us BY cs;

Step4: For each obtained in step 1, perform steps (5) thru (13).

Step5: Let NODES be a vector that represents all the K nodes in the cycle under consideration. For J = 1 to K perform steps (6) thru (12).

Step6: Let q = NODES(J). If SEQUENCED(q) > 0 then goto step 12.

Step7: Set SEQUENCED(q) = 1.

Step8: If q is contained in L, then output the compound node q, and the end-element. At end, goto step 12.

Step9: q contained in R. Therefore, find the boolean expression p that restricts the range of Is to the range (lq,uq,cq) which is the range associated with the node q.

Step10: Obtain the mapping function f that relates Iq and Is.

Step11: Output the do-element:

IF p THEN DO Iq = f(Is);

Step12: Continue.

Step13: Continue.

Step14: Output an end-element, and then return.

9.2.15.1 <u>Decomposition</u> <u>Criterion</u>(3).

If a cycle in an induced graph is sequenceable, but the sum of thresholds associated with that cycles is <= 0, then the graph is decomposed using the following criterion:

Pick any two nodes i and j in the cycle that are connected by a Ø2 edge, and for which the threshold is given by -T (where T is positive), and the map vector is given by:

$$M_{ij} = [l_x, u_x, c_x][l_y, u_y, c_y]$$

Now, apply one of the following decompositions:

(1). If $| c_y | > 1$ then decompose the node j (and all the related nodes) into two nodes k and l, the elements of which are given by:

$$D_k = l_j + 2.m.c_j \quad | \; 1 <= m <= n /2,$$

and $$D_l = D_j - D_k$$

where- (l_j, u_j, c_j) represents the set of elements represented by the node j and n is given by:

$$n = \lfloor (u_j - l_j) / c_j \rfloor$$

(2). If $| c_y | = 1$ then decompose node i (and all the related nodes) into the nodes represented by the following two sets of elements:

$$D_k = A(l_i + m.c_i) \quad | \quad 0 <= m <= T$$

$$D_l = D_i - D_k$$

where-

$A(l_i, u_i, c_i)$ represents the node i and D_i represents the set of elements represented by the node i.

In the following, the above decomposition is applied to the graph shown in Figure 9.16. Figure 9.17 shows the resulting graph after applying the the first part of the decomposition criterion described above to the node 2.

Figure 9.18 shows the resulting graph after applying, again, the first part of the above decomposition criterion to the nodes 21 and 4 of Figure 9.17. The decomposition criterion can be applied further and further to the graph of Figure 9.18 to obtain a graph in which the sum of thresholds on all Ø2 edges is positive, and then, get a schedule for the graph using the Algorithm A9.3.

9.2.16 Variables of Dimension Greater than 1.

We will now try to extend the results on array graph described in the earlier sections to the general case where the nodes represent variables of dimension > 1. Again, we will use the notation:

$$A(l_i^1, u_i^1, c_i^1)\ (l_i^2, u_i^2, c_i^2) \ldots (l_i^d, u_i^d, c_i^d)$$

to represent an array of dimension d. The subscript (i in the above example), associated with the bounds l, u, and c will be used as before to denote the node number. We will start with some additional definitions.

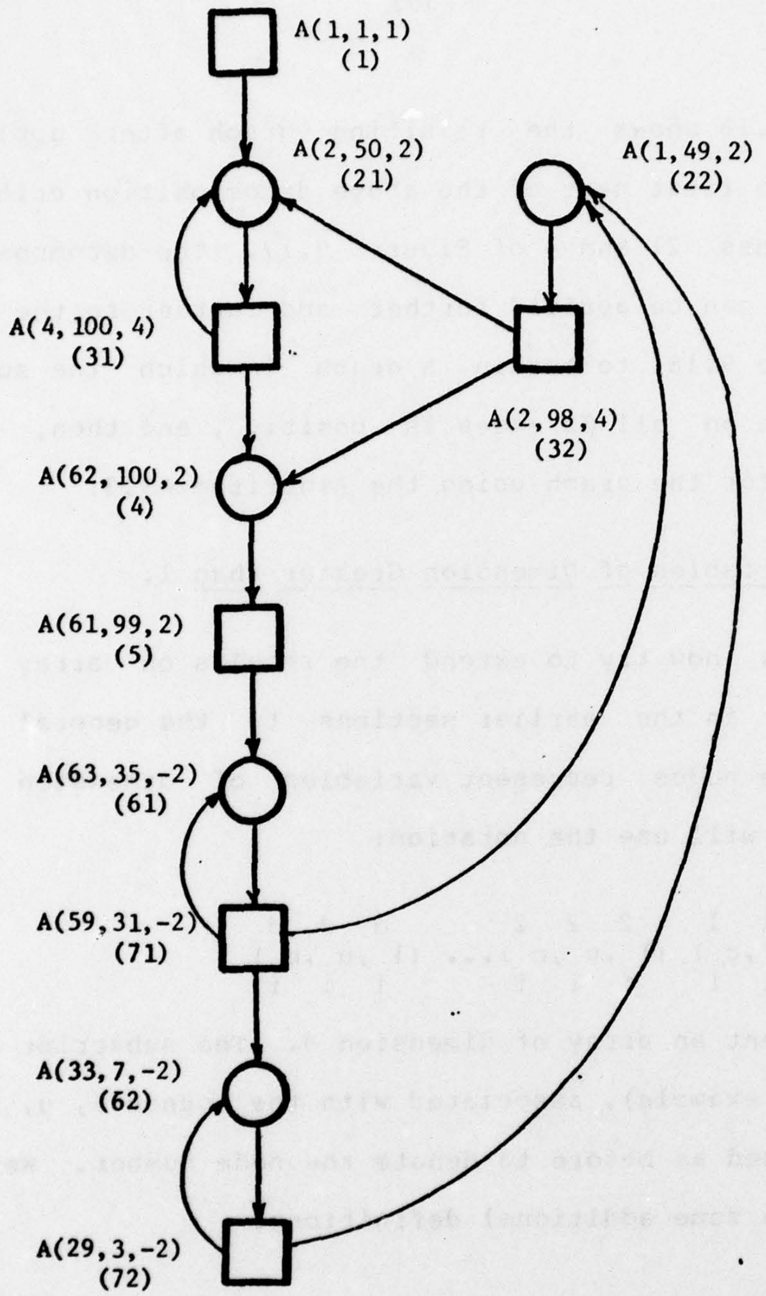


Figure 9.17. A Decomposition of Graph in Figure 9.16.

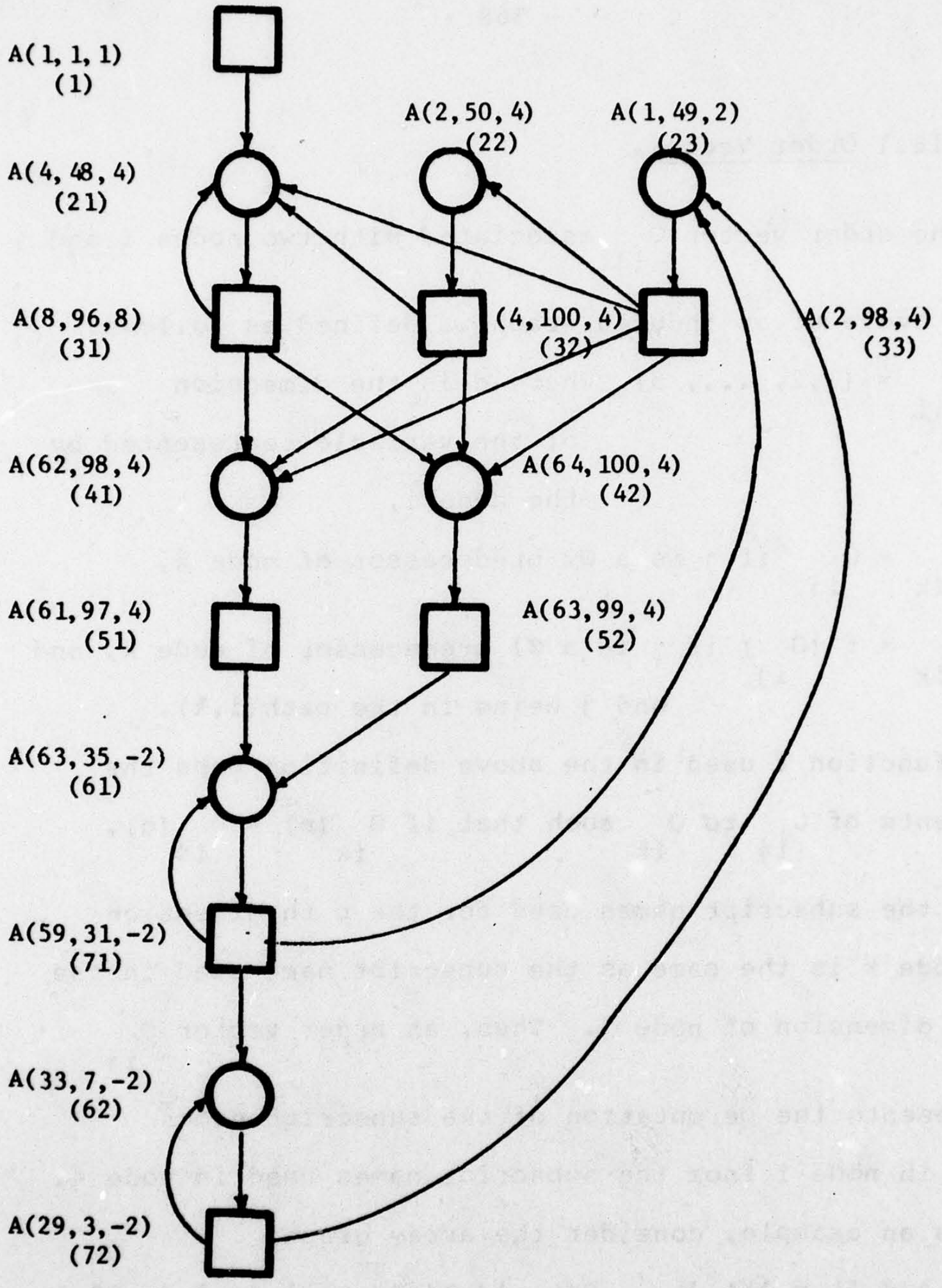


Figure 9.18. A Decomposition of Graph in Figure 9.17.

9.2.16.1 Order Vector.

The order vector O_{ij} associated with two nodes i and j in a cycle of an induced graph is defined as follows:

$O_{ii} = (1,2, \ldots, d)$ where d is the dimension of the variable represented by the node i,

$O_{ik} = O_{ij}$ if j is a Ø2 predecessor of node k,

$O_{ik} = f(O_{ij})$ if j is a Ø1 predecessor of node k, and and j being in the path(i,k).

The function f used in the above definition maps the elements of O_{ij} to O_{ik} such that if $O_{ik}(p) = O_{ij}(q)$, then the subscript names used for the p th dimension of node k is the same as the subscript name used in the q th dimension of node j. Thus, an order vector O_{ij} represents the permutation of the subscript names used in node i from the subscript names used in node j.

As an example, consider the array graph:

```
A(J,I) = A(I,J)     For  1<=I<=4  &  1 <= J <= 10
A(J-1,K) = A(K,J)   For  4<= J <= 9 &  4 <= K <= 10
```

The corresponding induced graph is shown in Figure 9.19. The subscript names used in the first and second dimensions of the node 1 are the same as the subscript names used in the second and the first dimensions of node 2. Therefore, we have $O_{12} = (2,1)$. Similarly, $O_{34} = (2,1)$. And, by definition, $O_{34} = (1,2)$.

9.2.16.2 Scale Factor.

Scale Factor F_i associated with the i th node of a directed graph is a set of d elements, where d is the dimension of the variable represented by the i th node and is defined recursively as follows:

```
    $f_i(d) = 1$

FOR k = d TO 2 BY -1

    $f_i(k-1) = f_i(k)*(1+(u_i^k - 1_i^k)/c_i^k)$
```

For example, for the directed graph shown in figure 8.6,

$$F_1 = (1,1) = F_4 = F_5$$

$$F_2 = (3,1) = F_3$$

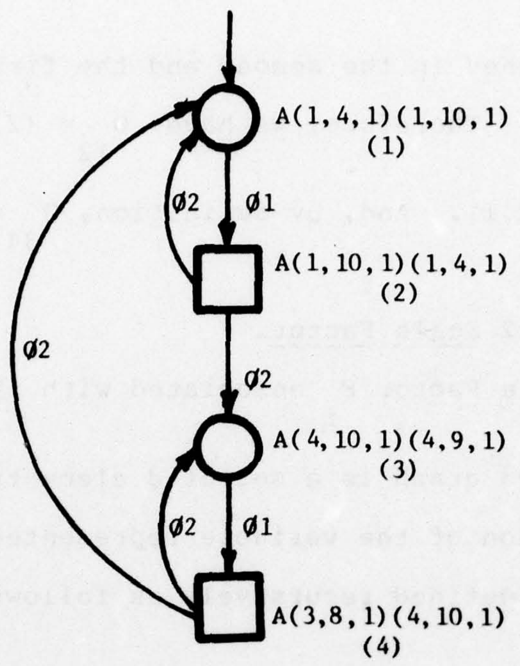


Figure 9.19. Induced Graph for the Assertions:

$A(J,I) = A(I,J)$ FOR $1 \leq I \leq 4$ & $1 \leq J \leq 10$

$A(J-1,K) = A(K,J)$ FOR $4 \leq J \leq 9$ & $4 \leq K \leq 10$

9.2.16.3 <u>Weight Vector</u>.

The Weight Vector W_{ij} associated with a Ø2 edge between two nodes i and j is defined as follows:

$$W_{ij} = F_j (O_{ij})$$

where-

O_{ij} is the order vector associated with the Ø2 edge,
and F_j is the Scale Factor associated with the node j.

For example, for the induced graph shown in Figure 8.6,

$$W_{32} = F_2 = (3,1)$$

and $$W_{52} = F_2 = (3,1)$$

9.2.16.4 <u>Compound Threshold</u>.

We have seen in section 9.2.15, that the threshold associated with a Ø2 edge of a cycle is the most useful tool in obtaining a condition for schedulability of the induced graph. The concept of threshold can be extended to the general case. We will call the corresponding threshold associated with the Ø2 edge of a general array graph as <u>Compound Threshold</u> (CT), and will define it as follows:

$$CT_{ij} = W^{1}_{ij} . T^{1}_{ij} + W^{2}_{ij} . T^{2}_{ij} + \ldots + W^{d}_{ij} . T^{d}_{ij}$$

where, T^{k}_{ij} is the threshold associated with the k th dimension, and which is defined the same way as in section 9.2.5; and $W^{1}_{ij}, \ldots, W^{d}_{ij}$ are the components of the weight vector associated with the Ø2 edge, and which is defined in the previous sub-section. For example, for the induced graph of Figure 8.6,

$$T^{1}_{32} = 0,$$

$$T^{2}_{32} = 1,$$

and therefore,

$$CT_{32} = 0 + 1 = 1.$$

Similarly,

$$CT_{52} = 1 * 3 + 0 = 3.$$

As in the simple case, the compound threshold also defines the distance between the closest elements which should be related by precedence.

9.2.16.5 Condition for Schedulability.

Again, we will extend the condition for schedulability, which was derived in section 9.2.15, to the general case. That is,

> An induced graph is schedulable, if the sum of compound thresholds associated with all the Ø2 edges is positive for every cycle in the graph.

We will not be describing the algorithm for scheduling a general array graph in this dissertation.

9.3 Consistency Analysis.

This section describes the consistency analysis performed in the MODEL Processor. As overviewed in Figure 9.3, consistency analysis consists essentially of the following 4 parts:

(1). Checking for multiple definition any data name or any element of a data name (Procedure RICONS8).

(2). Removing all the type-8* edges in the graph that cause cycle (Procedure REMOVE8).

(3). Checking the consistency of each recursion using the results described in the previous section (Procedure CHKSCON).

(4). Propagating the subscripts from the input nodes to the output nodes in the graph, at the same time checking for consistency of subscripts specified in the LHS and RHS of an assertion.

The Procedures that perform the above functions are described in the following.

9.3.1 Implementation Limitations.

Not all the results derived in section 9.2 are used in the implementation of the current version of the MODEL

---

* type-8 edges are the same as the Ø2 edges described in section 9.2.

processor. The items that are not included in the system are:

(1). Algorithms for splitting the assertions. The procedures ASPLIT1 and ASPLIT2 described in section 9.2.12 are not implemented. The algorithm for splitting the assertion using the decomposition criterion (3) is also not implemented. Therefore, if the condition given in section 9.2.15 is not satisfied, the graph is reported as not schedulable.

(2). The algorithm SCHEDUL described in the previous section is not implemented as it is part of the sequencing phase (see Chapter 11).

(3). If recursion is specified on more than one dimension of a variable, then the consistency check is not performed.

In addition to the above limitations, the consistency check is not performed when any of the lower bound, upper bound or increment associated with a node of the graph is not an integer.

9.3.2 <u>The Procedure RICONS8 (Report Inconsistencies)</u>.

This procedure checks to see if any data name or any element of a data name is defined more than once. The functions of this procedure can be described by the following steps.

(1). Using the routines G#TVARS and GTVARPS, obtain a list of all target variables for a data name.

(2). Using the routine MAPV, find the intersection between the set of elements represented by any two nodes in the list. If the intersection is non-empty, then some (or all) elements of the two nodes are multiply defined, so a warning message is sent to the user using the routine RICONSE.

9.3.3 The Procedure REMOVE8 (Remove Type-8 Edges).

During the creation of the Precedence Matrix, some type-8 relationships are established between different nodes of a data name. These type-8 edges can make the resulting graph cyclic if the elements of the data name is defined recursively. The sorting algorithms that are used later on require the graph to be acyclic, therefore, those type-8 edges that cause cycle must be removed. For example, both the directed graphs in Figures 9.5 and 9.6 are cyclic. The function of the REMOVE8 procedure is to remove the type-8 edges that cause cycles. Thus, the graphs in Figures 9.5 and 9.6 are changed to the graphs shown in Figures 9.20 and 9.21 respectively.

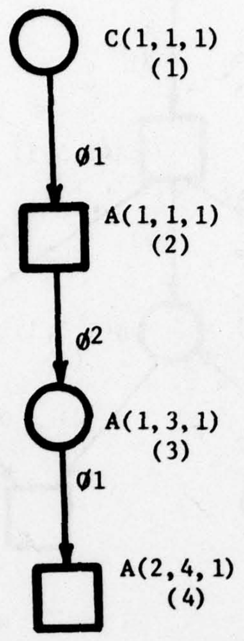


Figure 9.20. Resulting Graph after applying the REMOVEB Algorithm to the Induced Graph of Figure 9.5.

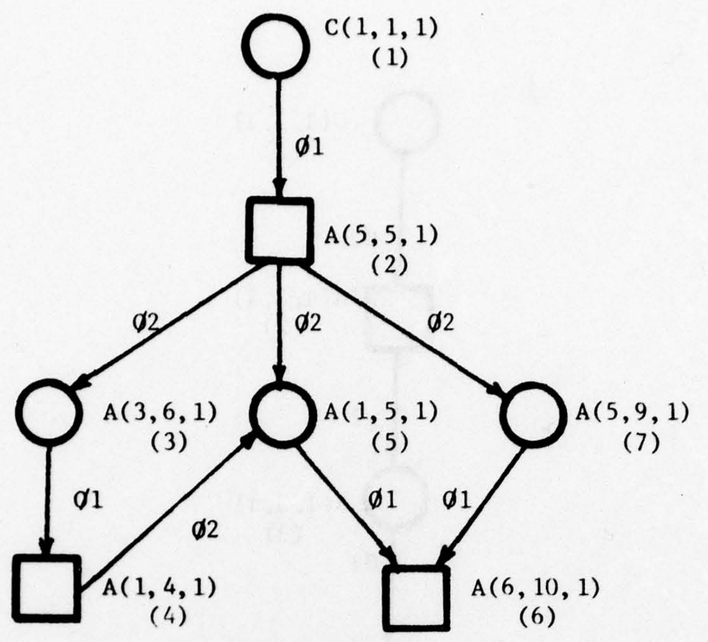


Figure 9.21. Resulting Graph after applying the REMOVE8 Algorithm to the Induced Graph of Figure 9.6.

Algorithm A9.4 describes the functions of the procedure REMOVE8. It takes the following vectors as input (where #DICT indicates the number of dictionary entries):

(1). #PRED - a vector of size #DICT, and the element #PRED(I) represents the number of predecessors of node I.

(2). #PRED8 - a vecotr of size #DICT, and the element #PRED8(I) represents the number of type-8 predecessors of node I.

(3). #SUCC8 - a vector of size #DICT, and the element #SUCC8(I) represents the number of type-8 successors of node I.

(4). PREDL - a vector of size n, where n = SUM(#PRED), and such that PREDL(#PREDF(I)) TO PREDL(#PREDT(I)) defines all the predecessors of node I.

(5). PRED8L - a vector of size n, where n = SUM(#PRED8), and such that PRED8L(#PRED8F(I)) TO PRED8L(#PRED8T(I)) defines all the type-8 predecessors of node I.

(6). SUCC8L - a vector of size n, where n = SUM(#SUCC8), and such that SUCC8L(#SUCC8F(I)) TO SUCC8L(#SUCC8T(I)) defines all the type-8 successors of node I.

(7). #PREDF, #PREDT, #PRED8F, #PRED8T, #SUCC8F, #SUCC8L, all of size #DICT, and define the predecessors, type-8 predecessors and type-8 successors as mentioned in (4), (5) and (6) above.

All the above vectors are computed in the procedure ORT#P, which is described in section 9.1.2.

As output, it generates the order-vector, ORDERV, which is of size #DICT and contains the ordered nodes.

Steps 1 thru 10 in the algorithm obtain the order vector using conventional topological sorting technique. Steps 12 thru 14 removes the type-8 edges that cause the cycle. If no type-8 edge is removed in these steps, then there must be some regular cycles in the graph (due to simultaneous equations, and this will be detected in the procedure CYCLES, described later on). In this case, a node that has least number of predecessors is assumed to be the next candidate for ordering (Step 15).

Algorithm A9.4: REMOVE8.

Function: Removes the type-8 edges that cause cycles.

Called by: RICONS.

Calls: REMOVE8, ADJM.

Inputs: #PRED, #PRED8, #SUCC8, PRED1, PRED8L, SUCC8L,

#PREDF, #PREDT, #PRED8F, #PRED8T, #SUCC8F,

```
          #SUCC8T.
     Outputs: ORDERV.
     Step1:  Set #ORDERED = 0 and T#PRED8 = #PRED8.
     Step2:  Set T#ORDERED = #ORDERED.
     Step3:  For I = 1 to #DICT perform steps 4 thru 10.
     Step4:  If #PRED(I) > 0 then goto step 10.
     Step5:  Do the following: #ORDERED = #ORDERED + 1; and
             ORDERV(#ORDERED) = I;
     Step6:  For each successor N of node I perform steps 7
             thru 9.
     Step7:  Set #PRED(N) = #PRED(N) - 1;
     Step8:  If ADJM(I,N) = 8 then set T#PRED8(N) =
             T#PRED8(N)-1;
     Step9:  Continue.
     Step10: Continue.
     Step11: If #ORDERED = #DICT then return; otherwise, if
             T#ORDERED < #ORDERED, then goto step 2, otherwise,
             continue to step 12.
     Step12: For I = 1 to #DICT perform steps 13 thru 14.
     Step13: If #PRED(I) = T#PRED8(I) and T#PRED8(I) <
             #PRED8(I) then remove every type-8 predecessor of
             node I using the routine REMOVE8 and then goto
             step 2.
     Step14: Continue.
```

Step15: Find a source variable node that has least number of predecessors, and for that node set #PRED(I) = 0, and then goto step 2.

9.3.4 <u>The Procedure CHKSCON (Check Subscript Consistency)</u>.

This procedure checks for the schedulability of the array graphs using the results obtained in section 9.2. The functions of the procedure are described in the following steps:

Step1: Using the Procedure GCYCLES, get all the cycles in the directed graph. Let N be the number of cycles obtained.

Step2: Let CONSISTENT be an array of N elements. Initialize all its elements to zeros.

Step3: For I = 1 to N perform steps 4 thru 8.

Step4: If CONSISTENT(I) > 0 then goto step 8.

Step5: Using the routine CTHRES, find the sum of thresholds associated with all the type-8 edges of the I th cycle obtained in step 1. If the sum is > 0 then set CONSISTENT(I) = 1 and goto step 8.

Step6: Using the routine CSMAPV, find the self map vector associated with every node of the I th cycle. If the condition in Theorem 9.1 is satisfied for this cycle, then set CONSISTENT(I) to 2, and goto step 8.

Step7: Using the Procedure CLINC, change the increment associated with a node (in which the computable elements are on right), and then apply the consistency criterions in steps (5) and (6) again.

Step8: Continue.

Step9: For I = 1 to N perform steps 10 thru 13.

Step10: If CONSISTENT(I) = 1 then goto step 13.

Step11: If CONSISTENT(I) = 2 then indicate that the I th cycle is sequenceable, but not schedulable.

Step12: If CONSISTENT(I) <=0 then indicate the error that the I th cycle is not sequenceable.

Step13: Continue.

9.3.5 <u>The Procedure SUBPROP</u> (<u>Subscript Propagation</u>).

The process of subscript propagation refers to the task of propagating subscript names and the ranges associated with the source variables of an assertion to the target variable of that assertion. It also checks for the consistency between the propagated subscript names (and ranges), and the specified subscript names (and ranges). The process is summarized in the following steps:

(1). Associate a subscript set with each dictionary entry that represents a field, interim or subscript name. This set contains a set of subscript names on which the entry depends. For example, for the dictionary entry that represents the subscripted data name: A(1,I,J), the set is (I,J).

(2). Obtain an order vector, ORDERV, that represents the topological sorting of the dictionary entries. And for each assertion node in ORDERV perform the following steps (ORDERV is needed in order to propagate subscript lists from the input nodes to the output nodes. Such propagation (from first entry to the last entry in ORDERV) allows us to resolve some inconsistencies in the subscript specification of some interim data names.):

(3). Find the set SR which is a union of the subscript names associated with all the source variables of the assertion. (If any of the source variables is used as an argument of a reduction function like, SUM, then the set associated with that name must be modified accodingly.) For example, if the RHS of an assertion is given by:

A(I)+B(J,I)+SUM(C(I,K))

then SR =(I,J), assuming that the SUM function is applied to the last dimension of C. Note that the order of subscript names is not maintained.

(4). Now compare the set SR with the set of subscript names SL associated with the target variable of the assertion. The subscript specification is consistent if the two sets satisfy one of the following criteria:

(a). SL = SR, or

(b). SL > SR, or

(c). SR < SL - if the variable on the LHS is an interim data name.

The second case is an example of an expansion function like:

```
X(I,J) = Y(J);
```

or like:

```
X(I) = 0;
```

In the third case, the data description statement of the interim data name on the LHS is modified to take care of the inconsistency.

(5). If the sets SL and SR do not satisfy one of the criterions mentioned in step (4), then an attempt is made to identify the subscript names in the set SL - {SL $\cap$ SR} with the subscript names in the set SR - {SL $\cap$ SR}. If any such identification does not cause conflicts (like, having same subscript name for two different source data names or two different target data names, as described in the procedure CSUBRED), then the corresponding subscript sets are changed, and the consistency check in step 4 is applied again. For example, consider the following assertion:

B(I) = A(J);

For this assertion, SL = {I}, and SR = {J}. Therefore, if identifying I with J does not cause any conflict, the assertion is changed to:

B(I) = A(I);

A conflict can occur when A and B are of different size.

(6). If an assertion contains more than one target variable, then the subscript set SL associated with each target variable must satisfy one of the criterions described in step (4).

The above functions are performed by the procedure SUBPROP. This procedure calls the routines: UNIONL, XNLIST, SUBTRL, UPDINTM, SUBPRC AND SUBPRU.

The routine UNIONL obtains the union of two arguement subscript lists. This is used to obtain the union of all the subscripts associated with the source variables of an assertion. The routine XNLIST is used to find the intersection of two argument lists and the routine SUBTRL is used to subtract a list from another.

The routine UPDINTM is used to update the dimension of an interim variable (see also, step (4) above), and to change the assertions that use it (routine UPDASSN is used for this purpose). The routine SUBPRC checks to see if any elements in the two arguments are equivalent. If two elements are equivalent, then one of the elements is deleted from the list.

The routine SUBPRU updates the data names pointed by POINTER varaibles. For example, if the following assertion was used in a specification:

POINTER.MAST = <expression>;

and the variable POINTER.MAST was recognized as an array of dimension n, then, n dimensions are added to the data name MAST. Also, the dimensions of all the successors of MAST, and their usage, are appropriately updated.

9.3.6 <u>Cycles Analysis</u> (<u>Procedure CYCLES</u>).

The user specification of a problem may be inconsistent in that a data name may be a predecessor of itself. In this case, there will be cycles in the directed graph. These cycles are detected using conventional techniques in the procedure CYCLES. The functions of the procedure

CYCLES are:

(1). To find all the strongly connected cycles in the graph.

(2). Check if there are simultaneous equations in the specification (among the strongly connected cycles obtained in step (1)). And, if there are, they are grouped together in the Associative Memory for later generation of a solution procedure.

(3). If the strongly connected cycles in the graph do not form a set of simultaneous equations, then, an error message is sent to the user requesting him to open the loops detected in the specification.

The detailed description of the CYCLES algorithm can be found in the dissertation by Gana (GAN78).

9.4 Some Examples.

In this section, two examples are given. These examples illustrate some of the functions performed by the MODEL Processor during the consistency and completeness analysis.

The first example is the MODEL specification shown in Figure 9.22. In this specification, some of the elements of array A are multiply defined and some of them are undefined. Therefore, as described earlier, the procedure

RESIC8 generates the assertions $A2001 and $A2002 shown in the bottom of the Figure 9.22. The warning report for this example, which is shown in Figure 9.23, indicates all the multiply defined elements of the array.

The second example is the MODEL specification in Figure 9.24. The induced graph for this example is shown in Figure 9.10. As mentioned earlier, the cycle in this graph is inconsistent, and because we do not split the assertions, the error message shown in Figure 9.25 is reported to the user.

```
                    SOURCE LISTING AND GENERATED STATEMENTS

STATEMENT
NUMBER
   1       FOR I = 1 TO 333 A(3*I+1) = A(I)+1;                                                  00000010
   1       FOR I = 1 TO 499 A(I) = A(2*I) + 1;                                                  00000020
   2       A(27) =0;                                                                            00000021
   3      /*****************************************************************************/ ********
   3      /*                                                                           */ ********
   3      /*   FOLLOWING STATEMENTS WERE GENERATED BY THE SYSTEM                       */ ********
   3      /*                                                                           */ ********
   3      /*****************************************************************************/ ********
   4      A IS INTERIM;                                                                         ********
   5      I IS SUBSCRIPT((1,*,1,1));                                                            ********
   6      $B1001 IS SUBSCRIPT((1,*,1,1));                                                       ********
   7      $B1002 IS SUBSCRIPT((1,*,1,1));                                                       ********
   8      $A2001:  FOR $B1002 =  504 TO 996 BY 6  A($B1002)=0;                                  ********
   9      $A2002:  FOR $B1001 =  500 TO 998 BY 6  A($B1001)=0;                                  ********
   9                                                                                            ********
   9                                                                                            ********
```

Figure 9.22.

```
WARNINGS.
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     1. ASSERTION NAME IS ASSUMED TO BE: "$A0001".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     2. ASSERTION NAME IS ASSUMED TO BE: "$A0002".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     3. ASSERTION NAME IS ASSUMED TO BE: "$A0003".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "I IS SUBSCRIPT((1,*,1,1));".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "$B1001 IS SUBSCRIPT((1,*,1,1));".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "$B1002 IS SUBSCRIPT((1,*,1,1));".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "$A2001:  FOR $B1002 =  504 TO 996 BY 6  A($B1002)=0;".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "$A2002:  FOR $B1001 =  500 TO 998 BY 6  A($B1001)=0;".
  THE FOLLOWING ELEMENTS: A(I),I = 4 TO 499 BY 3 ARE MULTIPLY DEFINED IN THE ASSERTIONS $A0001 AND $A0002
  THE FOLLOWING ELEMENTS: A(I),I = 27 TO 27 ARE MULTIPLY DEFINED IN THE ASSERTIONS $A0002 AND $A0003
```

Figure 9.23. Warnings for the MODEL Specification in Figure 9.22.

```
                    SOURCE LISTING AND GENERATED STATEMENTS

STATEMENT
NUMBER
   1      A(6) = 19;                                                                00000010
   2      FOR I = 5 TO 9  B(I) = A(I);                                              00000020
   2      FOR I = 6 TO 8  A(15-I) = B(I);                                           00000030
   3      /**********************************************************************/ ********
   3      /*                                                                    */ ********
   3      /*   FOLLOWING STATEMENTS WERE GENERATED BY THE SYSTEM                 */ ********
   3      /*                                                                    */ ********
   3      /**********************************************************************/ ********
   4      I IS SUBSCRIPT((1,*,1,1));                                                ********
   5      A IS INTERIM;                                                             ********
   6      B IS INTERIM;                                                             ********
```

Figure 9.24.

```
WARNINGS.
   NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     1. ASSERTION NAME IS ASSUMED TO BE: "$A0001".
   NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     2. ASSERTION NAME IS ASSUMED TO BE: "$A0002".
   NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER     3. ASSERTION NAME IS ASSUMED TO BE: "$A0003".
   THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "I IS SUBSCRIPT((1,*,1,1));".


ERRORS.
   THE LOOP CONSISTING OF THE FOLLOWING NODES IS INCONSISTENT: (10,2,9,8,1,5).  THE CORRESPONDING NODE NAMES ARE: (A,$A0002,B,B,
         $A0003,A).
```

Figure 3.25. Warnings and ERRORS for the MODEL Specification in Figure 9.24.

CHAPTER 10

DOCUMENTATION

This chapter describes the fourth phase of the MODEL Processor, namely, the Documentation Phase. The main function of this phase is to generate the required user reports. Some of the reports are generated throughout the first 4 phases. Therefore, all the documentation produced by the first four phases of the Processor will be described (including the different error reports) in this chapter.

Basically, the MODEL Processor generates 4 user reports. They are:

(1). Source Listing (SAPOUT),

(2). Cross Reference Report (XREF),

(3). Formatted MODEL Specification Listing (MODELOUT),

and (4). Precedence Matrix Report (SYSPRINT).

The MODEL Processor also generates 3 kinds of error reports. They are:

(1). Syntax error report (SAPERR),

(2). Semantic error report (ERROR),

and (3). Warnings (WARNING).

The names inside parantheses in the above lists indicate the file names to which the corresponding report is output.

Figure 10.1 shows the time at which each of the above reports are generated. In the following, the generation of different reports is described briefly. The example in Figure 4.5 is used to illustrate the different reports.

10.1 Source Listing.

This report consists of the input MODEL statements and all the statements generated by the MODEL Processor. This report is generated throughout the first four phases of the Processor. Each entry in the report is identified by the statement number on the left. Every other report refers to a statement by referring to this statement number. On the RHS of each statement, the user supplied line number is indicated. If no line number was specified in the user input statement, then this represents the sequence number of that statement. If the statement was generated by the system, this entry will be replaced by '*********'.

The source listing for the MODEL specification of Figure 4.5 is given in Figure 10.2. This listing is output to the file SAPOUT, and the procedure WSAPOUT is used to output a line to this file.

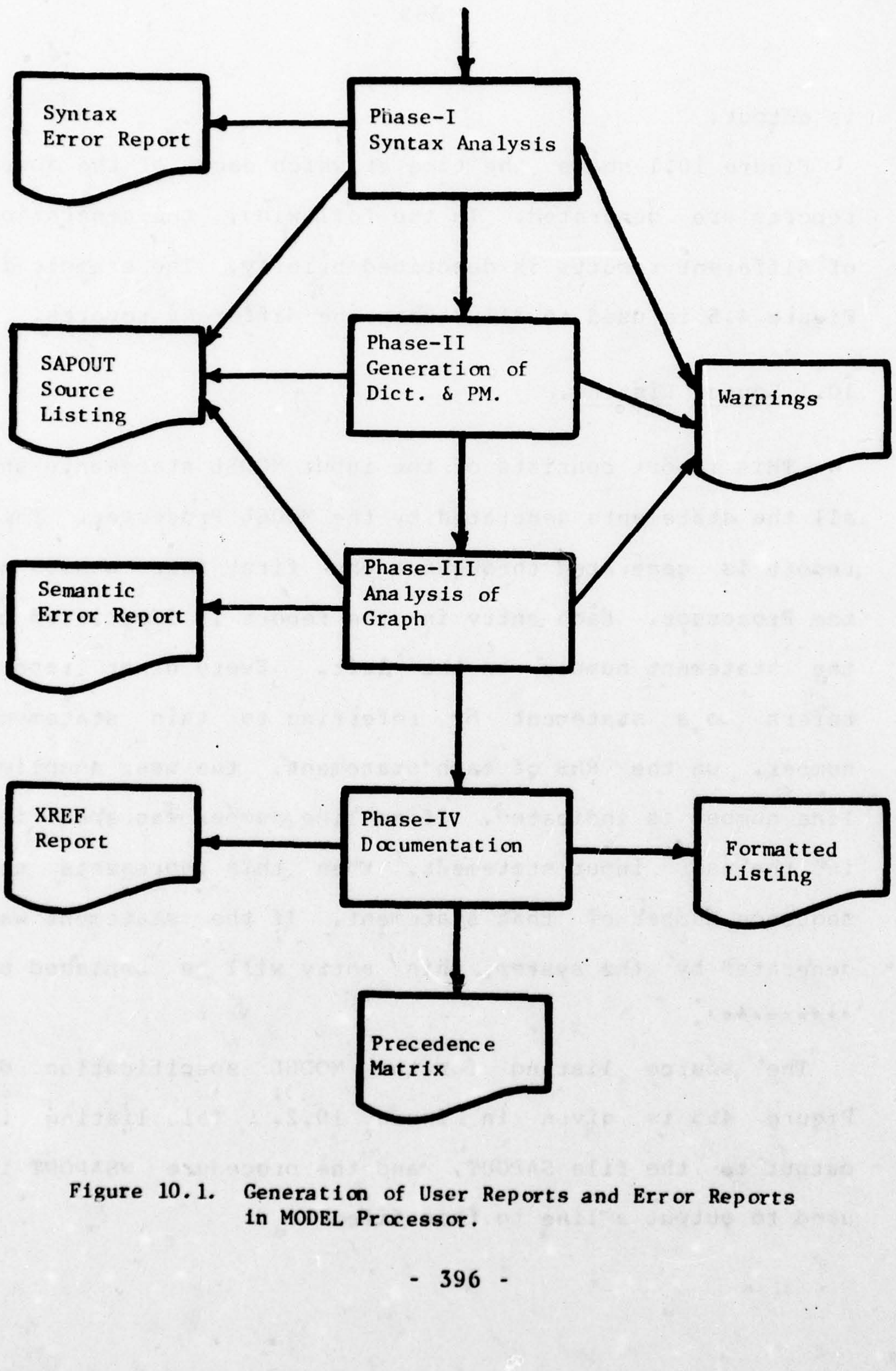


Figure 10.1. Generation of User Reports and Error Reports in MODEL Processor.

SOURCE LISTING AND GENERATED STATEMENTS

```
STATEMENT
NUMBER
    1      MODULE: RUN_1:                                             00000010
    2       SOURCE: TRANSACTION.MASTER:                               00000020
    3       TARGET: REPORT_1.MASTER:                                  00000030
    4       TAPE IS MEDIA(TAPE):                                      00000040
    5       MASTER IS FILE(TAPE.SAM.KEY=PRODUCT_NUMBER):              00000050
    6       MAST IS RECORD(MASTER.(*)):                               00000060
    7       PRODUCT_NUMBER IS FIELD(MAST.PIC'X(5)'):                  00000070
    8       UNIT_PRICE IS FIELD(MAST.PIC'S99V99'):                    00000080
    9       YTD_SALES IS FIELD(MAST.PIC'9(5)V99'):                    00000090
   10       FILLER IS FIELD(MAST.CHAR(3)):                            00000100
   11                                                                 00000110
   11      TRANSACTION IS FILE(TAPE.SAM.KEY=PRODUCT_NUMBER):          00000120
   12       PRODUCT IS GROUP(TRANSACTION.(*)):                        00000130
   13       TRANS IS RECORD(PRODUCT.(*)):                             00000140
   14       PRODUCT_NUMBER IS FIELD(TRANS.PIC'X(5)'):                 00000150
   15       QUANTITY IS FIELD(TRANS.PIC'S99'):                        00000160
   16       SALESMAN IS FIELD(TRANS.PIC'XX'):                         00000170
   17       DISTRICT IS FIELD(TRANS.PIC'A'):                          00000180
   18       FILLER IS FIELD(TRANS.CHAR(69)):                          00000190
   19                                                                 00000200
   19      PRINTER IS MEDIA(PRINT):                                   00000210
   20       REPORT_1 IS FILE(PRINTER.SAM.KEY=PRODUCT_NUMBER):         00000220
   21       PAGE IS GROUP(REPORT_1):                                  00000230
   22       HEADING IS RECORD(PAGE):                                  00000240
   23       PRODUCT_ID IS FIELD(HEADING.PIC'X(10)'):                  00000250
   24       PRODUCT_TOTAL IS FIELD(HEADING.PIC'X(14)'):               00000260
   25       PRODUCT_YTD IS FIELD(HEADING.PIC'X(14)'):                 00000270
   26       REP_1 IS RECORD(PAGE.(*)):                                00000280
   27       FILLER1 IS FIELD(REP_1.PIC'X(5)'):                        00000290
   28       PRODUCT_NUMBER IS FIELD(REP_1.PIC'X(5)'):                 00000300
   29       FILLER IS FIELD(REP_1.PIC'X(5)'):                         00000310
   30       PRODUCT_TOTAL_FOR_MONTH IS FIELD(REP_1.PIC'ZZZZZ.99'):    00000320
   31       YTD_SALES IS FIELD(REP_1.PIC'ZZZZZ.99'):                  00000330
   32       J IS SUBSCRIPT_(TRANS):                                   00000340
   33                                                                 00000350
   33       PRODUCT_TOTAL_FOR_MONTH = SUM(QUANTITY(J)*UNIT_PRICE):    00000360
   34        TARGET.YTD_SALES = SOURCE.YTD_SALES+PRODUCT_TOTAL_FOR_MONTH:  00000370
   35       POINTER.MAST = ANY( PRODUCT.PRODUCT_NUMBER):              00000380
   36       PRODUCT_ID = 'PRODUCT':                                   00000390
   37       PRODUCT_TOTAL = 'MONTH SALES':                            00000400
   38       PRODUCT_YTD = 'YTD SALES':                                00000410
   39       IF TRANS.PRODUCT_NUMBER(J) ¬= TRANS.PRODUCT_NUMBER(J+1) THEN   00000420
   39         ENDGRP.TRANS = TRUE                                     00000430
   39         ELSE ENDGRP.TRANS = FALSE                               00000440
   39      /***********************************************************/ ********
   39      /*                                                         */ ********
   39      /*   FOLLOWING STATEMENTS WERE GENERATED BY THE SYSTEM     */ ********
   39      /*                                                         */ ********
   39      /***********************************************************/ ********
   40      EXIST IS GROUP:                                            ********
   41      PRINTER IS GROUP(EXIST):                                   ********
   42      REPORT_1 IS GROUP(PRINTER):                                ********
   43      PAGE IS GROUP(REPORT_1):                                   ********
   44      REP_1 IS INTERIM(PAGE):                                    ********
   45      TAPE IS GROUP(EXIST):                                      ********
   46      TRANSACTION IS GROUP(TAPE):                                ********
```

```
47     PRODUCT IS GROUP(TRANSACTION);                                              ********
48     TRANS IS INTERIM(PRODUCT);                                                  ********
49     MASTER IS GROUP(TAPE);                                                      ********
50     MAST IS INTERIM(MASTER);                                                    ********
51     ENDGRP IS GROUP;                                                            ********
52     TRANS IS FIELD(ENDGRP);                                                     ********
53     POINTER IS GROUP;                                                           ********
54     MAST IS FIELD(POINTER);                                                     ********
55     $A$10001: $10001=QUANTITY(J) * UNIT_PRICE;                                  ********
56     $10001 IS INTERIM;                                                          ********
57     $A20001: PRINTER.REPORT_1.PAGE.REP_1.YTD_SALES=TAPE.MASTER.MAST.YTD_SALES********
58     $A20002: PRINTER.REPORT_1.PAGE.REP_1.FILLER=0;                              ********
59     $A20003: PRINTER.REPORT_1.PAGE.REP_1.PRODUCT_NUMBER=ANY(TAPE.TRANSACTION.********
59     PRODUCT.TRANS.PRODUCT_NUMBER);                                              ********
60     $A20004: PRINTER.REPORT_1.PAGE.REP_1.FILLER1=0;                             ********
61     $B10001 IS SUBSCRIPT((1,*,1,1));                                            ********
62     $B10002 IS SUBSCRIPT((1,*,1,1));                                            ********
63     $B10003 IS SUBSCRIPT((1,*,1,1));                                            ********
64     $B10004 IS SUBSCRIPT((1,*,1,1));                                            ********
```

Figure 10.2. Source Listing for the MODEL Specification in Figure 4.10.

10.2 The Cross Reference Report.

This report consists of an entry for each data name or an assertion name used in the MODEL specification. The different entries are listed in alphabetical order by the name. With each entry, the following information is provided:

(a). Statement number in which the name is defined (same as the statement number on the LHS of each entry in Source Listing described in section 10.1).

(b). The user supplied line number associated with the name. If the statement for the name was generated by the system, then this entry will be replaced by '*******'.

(c). The dictionary index for the name.

(d). Attributes of the name, like, size, parent data names, type of data name, and so on.

(e). A list of statement numbers that refer this data name.

The Cross Reference Report for the RUN_1 example in Figure 4.5 is shown in Figure 10.3. This report is output to the file XREF, and the procedures BUILDD and XREFPRT are used to generate this report. The procedure BUILDD builds a dictionary of data names from the directory of the Associative Memory. The procedure XREFPRT generates the report for each entry in the dictionary created by the procedure BUILDD.

10.3 The Formatted MODEL Specification Listing.

This report consists of the formatted listing of the MODEL specification. It consists of three sections:

(1). Header Section,

(2). Data description Section,

and (3). Assertions Section.

The statements in the data descriptions section are indented appropriately to reflect the tree structure of the files and interim data names. The assertions section is divided into two parts. The first part consists of simple assertions and the second part consists of simultaneous set of equations, if any.

The formatted listing for the RUN_1 example of Figure 4.5 is shown in Figure 10.4. This listing is output to the file MODELOUT. The procedures PRTMED, PRTSTMT and PRTASSX are used to generate this listing. PRTSTMT is used to list each data description statement and is called by the procedure PRTMED. The procedure PRTASSX is used to list all the assertions in the MODEL specification.

```
                                        ATTRIBUTES AND CROSS-REFERENCE TABLE
STMT NO. DCL NO. DICT NO. IDENTIFIER                         ATTRIBUTE AND REFERENCES

  55     ******  63       $ASI0001                           ASSERTION NAME
  33     360     35       $ACC01                             ASSERTION NAME
  34     370     34       $ACC02                             ASSERTION NAME
  35     380     33       $ACC03                             ASSERTION NAME
  36     390     32       $AC004                             ASSERTION NAME
  37     400     31       $ACC05                             ASSERTION NAME
  38     410     30       $AC006                             ASSERTION NAME
  39     440     29       $ACC07                             ASSERTION NAME
  63     ******  84       FOREACH$MAST                       (1,EXIST.TAPE.MASTER.MAST,1,1)IN MAST(EXIST.TAPE.MASTER.MAST) IN MASTER IN
                                                             TAPE, SUBSCRIPT
                                                             34
  64     ******  85       FOREACH$REP_1                      (1,EXIST.PRINTER.REPORT_1.PAGE.REP_1,1,1)IN REP_1(EXIST.PRINTER.REPORT_1
                                                             PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER, SUBSCRIPT
                                                             33,55,57,58,59,60
  56     ******  65       $IC001                             (*,*) INTERIM
                                                             33,55,56
  17     180     21       DISTRICT                           IN TRANS(EXIST.TAPE.TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE
                                                             TRANSACTION.PRODUCT) IN TRANSACTION IN TAPE, FIELD(PICTURE 'X')
  51     ******  50       ENDGRP                             GROUP
                                                             52
  40     ******  36       EXIST                              GROUP
                                                             41,45
  29     310     14       FILLER                             IN REP_1(EXIST.PRINTER.REPORT_1.PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER
                                                              FIELD(PICTURE 'X(5)')
                                                             29,58
  18     190     20       FILLER                             IN TRANS(EXIST.TAPE.TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE
                                                             TRANSACTION.PRODUCT) IN TRANSACTION IN TAPE, FIELD(CHAR(69))
  10     100     25       FILLER                             IN MAST(EXIST.TAPE.MASTER.MAST) IN MASTER IN TAPE, FIELD(CHAR(63))
  27     290     16       FILLER1                            IN REP_1(EXIST.PRINTER.REPORT_1.PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER
                                                              FIELD(PICTURE 'X(5)')
                                                             27,60
  22     240     8        HEADING                            IN PAGE IN REPORT_1 IN PRINTER, RECORD
                                                             23,24,25
  32     340     40       J                                  (1,EXIST.TAPE.TRANSACTION.PRODUCT.TRANS,1,1)IN TRANS(EXIST.TAPE
                                                             TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE.TRANSACTION.PRODUCT) IN
                                                             TRANSACTION IN TAPE, SUBSCRIPT
                                                             39,55
```

```
54     ****** 57    MAST                       (*)IN POINTER, FIELD
                                               6,35,54
50     ****** 46    MAST                       IN MASTER IN TAPE IN EXIST, INTERIM
                                               6
--------------------------------------------------------------------------------------------------------
6      60     11    MAST                       (EXIST.TAPE.MASTER.MAST)IN MASTER IN TAPE, RECORD
                                               7,8,9,10
5      50     5     MASTER                     IN TAPE, FILE
                                               6
--------------------------------------------------------------------------------------------------------
49     ****** 45    MASTER                     IN TAPE IN EXIST, GROUP
                                               50
43     ****** 39    PAGE                       IN REPORT_1 IN PAINTER IN EXIST, GROUP
                                               44
--------------------------------------------------------------------------------------------------------
21     230    7     PAGE                       IN REPORT_1 IN PRINTER, GROUP
                                               22,26
53     ****** 56    POINTER                    GROUP
                                               54
--------------------------------------------------------------------------------------------------------
41     ****** 37    PRINTER                    IN EXIST, GROUP
                                               42
19     210    1     PRINTER                    MEDIA
                                               20
--------------------------------------------------------------------------------------------------------
47     ****** 43    PRODUCT                    IN TRANSACTION IN TAPE IN EXIST, GROUP
                                               48
12     130    10    PRODUCT                    (EXIST.TAPE.TRANSACTION.PRODUCT)IN TRANSACTION IN TAPE, GROUP
                                               13
--------------------------------------------------------------------------------------------------------
23     250    19    PRODUCT_ID                 IN HEADING IN PAGE IN REPORT_1 IN PRINTER, FIELD(PICTURE 'X(10)')
                                               23,36
28     300    15    PRODUCT_NUMBER             IN REP_1(EXIST.PRINTER.REPORT_1.PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER
                                                FIELD(PICTURE 'X(5)')
--------------------------------------------------------------------------------------------------------
                                               28,59
14     150    24    PRODUCT_NUMBER             IN TRANS(EXIST.TAPE.TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE
                                               TRANSACTION.PRODUCT) IN TRANSACTION IN TAPE, FIELD(PICTURE 'X(5)')
                                               14,14,14,35,39,39,59
--------------------------------------------------------------------------------------------------------
7      70     28    PRODUCT_NUMBER             IN MAST(EXIST.TAPE.MASTER.MAST) IN MASTER IN TAPE, FIELD(PICTURE 'X(5)')
24     260    18    PRODUCT_TOTAL              IN HEADING IN PAGE IN REPORT_1 IN PRINTER, FIELD(PICTURE 'X(14)')
                                               24,37
30     320    13    PRODUCT_TOTAL_FOR_MONTH    IN REP_1(EXIST.PRINTER.REPORT_1.PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER
                                                FIELD(PICTURE 'ZZZZZ.99')
                                               30,33,34
25     270    17    PRODUCT_YTD                IN HEADING IN PAGE IN REPORT_1 IN PRINTER, FIELD(PICTURE 'X(14)')
                                               25,38
--------------------------------------------------------------------------------------------------------
15     160    23    QUANTITY                   IN TRANS(EXIST.TAPE.TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE
                                               TRANSACTION.PRODUCT) IN TRANSACTION IN TAPE, FIELD(PICTURE '999')
                                               15,55
```

```
44    ******  40   REP_1          IN PAGE IN REPORT_1 IN PRINTER IN EXIST, INTERIM
                                  26

26    280     6    REP_1          (EXIST.PRINTER.REPORT_1.PAGE.REP_1)IN PAGE IN REPORT_1 IN PRINTER, RECORD
                                  27,28,29,30,31
-----------------------------------------------------------------------------------------------------
20    220     3    REPORT_1       IN PRINTER, FILE
                                  21

42    ******  38   REPORT_1       IN PRINTER IN EXIST, GROUP
                                  43

1     10      0    RUN_1          MODULE NAME

16    170     22   SALESMAN       IN TRANS(EXIST.TAPE.TRANSACTION.PRODUCT.TRANS) IN PRODUCT(EXIST.TAPE
                                  TRANSACTION.PRODUCT) IN TRANSACTION IN TAPE, FIELD(PICTURE 'XX')

45    ******  41   TAPE           IN EXIST, GROUP
                                  46,49

4     40      2    TAPE           MEDIA
                                  5,11
-----------------------------------------------------------------------------------------------------
52    ******  51   TRANS          (*,*)IN ENDGRP, FIELD
                                  39,52

48    ******  44   TRANS          IN PRODUCT IN TRANSACTION IN TAPE IN EXIST, INTERIM
                                  13

13    140     9    TRANS          (EXIST.TAPE.TRANSACTION.PRODUCT.TRANS)IN PRODUCT(EXIST.TAPE.TRANSACTIC.
                                  PRODUCT) IN TRANSACTION IN TAPE, RECORD
                                  14,15,16,17,18

11    120     4    TRANSACTION    IN TAPE, FILE
                                  12

46    ******  42   TRANSACTION    IN TAPE IN EXIST, GROUP
                                  47
-----------------------------------------------------------------------------------------------------
8     80      27   UNIT_PRICE     IN MAST(EXIST.TAPE.MASTER.MAST) IN MASTER IN TAPE, FIELD(PICTURE '999V99')
                                  6,55

31    330     12   YTD_SALES      IN REP_1(EXIST.PRINTER.REPORT_1.PAGE.REP_1) IN PAGE IN REPORT_1 IN PRINTER
                                   FIELD(PICTURE 'ZZZZZ.99')
                                  31,57
-----------------------------------------------------------------------------------------------------
9     90      26   YTD_SALES      IN MAST(EXIST.TAPE.MASTER.MAST) IN MASTER IN TAPE, FIELD(PICTURE '9(5)V99'
                                  9,34,34,57
```

Figure 10.3. Cross Reference Report for the MODEL Specification in Figure 4.10.

```
/*****************************************************************/
/*                                                               */
/*                  RUN_1 MODULE DESCRIPTION                     */
/*                                                               */
/*****************************************************************/

MODULE: RUN_1:
SOURCE FILES: TRANSACTION,MASTER:
TARGET FILES: REPORT_1,MASTER:

/*****************************************************************/
/*                                                               */
/*               DATA DESCRIPTION STATEMENTS                     */
/*                                                               */
/*****************************************************************/

  PRINTER IS MEDIA(UNIT=TERMINAL);
  TAPE IS MEDIA(UNIT=TAPE);

  /***************************************************************/
  /*                                                             */
  /*              "REPORT_1" FILE DESCRIPTION                    */
  /*                                                             */
  /***************************************************************/

    REPORT_1 IS FILE(PRINTER,KEY=PRODUCT_NUMBER,SAM);
      PAGE IS GROUP(REPORT_1);
        HEADING IS RECORD(PAGE);
          PRODUCT_ID IS FIELD(HEADING,PICTURE 'X(10)');
          PRODUCT_TOTAL IS FIELD(HEADING,PICTURE 'X(14)');
          PRODUCT_YTD IS FIELD(HEADING,PICTURE 'X(14)');
        REP_1 IS RECORD(PAGE,(EXIST,PRINTER,REPORT_1,PAGE,REP_1));
          FILLER1 IS FIELD(REP_1,PICTURE 'X(5)');
          PRODUCT_NUMBER IS FIELD(REP_1,PICTURE 'X(5)');
          FILLER IS FIELD(REP_1,PICTURE 'X(5)');
          PRODUCT_TOTAL_FOR_MONTH IS FIELD(REP_1,PICTURE 'ZZZZZ.99');
          YTD_SALES IS FIELD(REP_1,PICTURE 'ZZZZZ.99');

  /***************************************************************/
  /*                                                             */
  /*            "TRANSACTION" FILE DESCRIPTION                   */
  /*                                                             */
  /***************************************************************/

    TRANSACTION IS FILE(TAPE,KEY=PRODUCT_NUMBER,SAM);
      PRODUCT IS GROUP(TRANSACTION,(EXIST,TAPE,TRANSACTION,PRODUCT));
        TRANS IS RECORD(PRODUCT,(EXIST,TAPE,TRANSACTION,PRODUCT,TRANS));
          PRODUCT_NUMBER IS FIELD(TRANS,PICTURE 'X(5)');
          QUANTITY IS FIELD(TRANS,PICTURE '999');
          SALESMAN IS FIELD(TRANS,PICTURE 'XX');
          DISTRICT IS FIELD(TRANS,PICTURE 'X');
          FILLER IS FIELD(TRANS,CHAR(65));

  /***************************************************************/
  /*                                                             */
  /*               "MASTER" FILE DESCRIPTION                     */
  /*                                                             */
  /***************************************************************/

    MASTER IS FILE(TAPE,KEY=PRODUCT_NUMBER,SAM);
      MAST IS RECORD(MASTER,(EXIST,TAPE,MASTER,MAST));
        PRODUCT_NUMBER IS FIELD(MAST,PICTURE 'X(5)');
        UNIT_PRICE IS FIELD(MAST,PICTURE '99V99');
        YTD_SALES IS FIELD(MAST,PICTURE '9(5)V99');
        FILLER IS FIELD(MAST,CHAR(63));
/*************************************************************************/
/*                                                                       */
/*                INTERIM DATA DESCRIPTION STATEMENTS                    */
/*                                                                       */
/*************************************************************************/
     POINTER IS GROUP;
       MAST IS INTERIM(POINTER,(*));
     ENDGRP IS GROUP;
       TRANS IS INTERIM(ENDGRP,(*,*));
     EXIST IS GROUP;
       TAPE IS GROUP(EXIST);
         MASTER IS GROUP(TAPE);
           PAST IS INTERIM(MASTER);
         TRANSACTION IS GROUP(TAPE);
           PRODUCT IS GROUP(TRANSACTION);
             TRANS IS INTERIM(PRODUCT);
       PRINTER IS GROUP(EXIST);
         REPORT_1 IS GROUP(PRINTER);
           PAGE IS GROUP(REPORT_1);
             REP_1 IS INTERIM(PAGE);
     $10001 IS INTERIM((*,*));
/*************************************************************************/
/*                                                                       */
/*                SUBSCRIPT DESCRIPTION STATEMENTS                       */
/*                                                                       */
/*************************************************************************/
     FOREACHSREP_1 IS SUBSCRIPT(REP_1,(1,
       EXIST.PRINTER.REPORT_1.PAGE.REP_1,1,1));
     FOREACHSMAST IS SUBSCRIPT(MAST,(1,EXIST.TAPE.MASTER.MAST,1,1));
     J IS SUBSCRIPT(TRANS,(1,EXIST.TAPE.TRANSACTION.PRODUCT.TRANS,1,1));
/*************************************************************************/
/*                                                                       */
/*                        ASSERTIONS                                     */
/*                                                                       */
/*************************************************************************/
    $AS10001:
      $10001(J,FOREACHSREP_1)=QUANTITY(FOREACHSREP_1,J) *
        SOURCE.UNIT_PRICE(FOREACHSREP_1);
    $ACC07:
      IF TAPE.TRANSACTION.PRODUCT.PRODUCT_NUMBER(FOREACHSREP_1,J) ¬=
        TAPE.TRANSACTION.PRODUCT.PRODUCT_NUMBER(FOREACHSREP_1,J + 1)
        THEN ENDGRP.TRANS(J,FOREACHSREP_1)='1'B;
      ELSE ENDGRP.TRANS(J,FOREACHSREP_1)='0'B;
    $ACC03:
      POINTER.PAST(FOREACHSREP_1)=ANY(
        TAPE.TRANSACTION.PRODUCT.PRODUCT_NUMBER(FOREACHSREP_1,J));
    $ACC02:
      TARGET.TAPE.MASTER.MAST.YTD_SALES(FOREACHSMAST)=
        SOURCE.TAPE.MASTER.MAST.YTD_SALES(FOREACHSMAST) +
        PRODUCT_TOTAL_FOR_MONTH(FOREACHSPAST);
```

```
                                                                  00004727
SA0004:                                                           00004728
  PRODUCT_ID='PRODUCT';                                           00004729
                                                                  00004730
SA0005:                                                           00004731
  PRODUCT_TOTAL='MONTH SALES';                                    00004732
                                                                  00004733
SA0006:                                                           00004734
  PRODUCT_YTD='YTD SALES';                                        00004735
                                                                  00004736
SA0001:                                                           00004737
  PRODUCT_TOTAL_FOR_MONTH(FOREACH$REP_1)=SUM($I0001(J,FOREACH$REP_1));00004738
```

Figure 10.4. Formatted MODEL specification for the MODEL Specification in Figure 4.10.

10.4 The Precedence Matrix Report.

This reports consists of a listing of all the relationships in the Precedence Matrix. The listing is not in the form of a matrix, but in the form of a list. A list of all predecessor nodes and the successor nodes of each entry in the precedence matrix is output on each line. The actual relationship itself (that is, the precedence matrix entry) follows each predecessor and successor node, by appropriately enclosing it in parantheses.

The Precedence Matrix Report for the RUN-1 example is shown in Figure 10.5. This report is output to the file SYSPRINT, and the procedure PRTADJL is used for generating this report. If required, the matrix form of the report can also be obtained using the procedure PRTADJM. When the number of rows in the precedence matrix exceed 50, it is easy to understand the List form of the report than the matrix form, and therefore, only the list form of the Precedence Matrix is illustrated here.

```
                                    PRECEDENCE MATRIX

DICT#  SUCCESSORS                                        PRECECESSORS

  1                                                       3(2)
  2    4(1)   5(1)
  3    1(2)                                               7(2)
  4   10(1)                                               2(1)   5(9)
  5   11(1)   4(9)                                        2(1)
  6    7(2)                                              12(2)  13(2)  14(2)  15(2)  16(2)  40(6)  8(9)
  7    3(2)                                               6(2)   8(2)
  8    7(2)   6(9)                                       17(2)  18(2)  19(2)
  9   20(1)  21(1)  22(1)  23(1)  24(1)                  10(1)  44(5)
 10    9(1)                                               4(1)
 11   25(1)  26(1)  27(1)  28(1)                          5(1)  46(6)  57(7)
 12    6(2)                                              75(2)  13(9)
 13    6(2)  12(9)                                       64(2)  14(9)
 14    6(2)  13(9)                                       77(2)  15(9)
 15    6(2)  14(9)                                       79(2)  16(9)
 16    6(2)  15(9)                                       81(2)
 17    8(2)                                              53(2)  18(9)
 18    8(2)  17(9)                                       54(2)  19(9)
 19    8(2)  18(9)                                       55(2)
 20                                                       9(1)  21(9)
 21   20(9)                                               9(1)  22(9)
 22   21(9)                                               9(1)  23(9)
 23   68(1)  22(9)                                        9(1)  24(9)
 24   47(1)  49(1)  59(1)  23(9)                          9(1)
 25                                                      11(1)  26(9)
 26   61(1)  25(9)                                       11(1)  27(9)
 27   69(1)  26(9)                                       11(1)  28(9)
 28   27(9)                                              11(1)
 29   52(4)                                              47(3)  48(3)  49(3)  82(3)
 30   53(4)
 31   54(4)
 32   55(4)
 33   58(4)                                              59(3)  83(3)
 34   60(4)                                              61(3)  62(3)  84(3)
 35   64(4)                                              66(3)  85(3)
 36                                                      37(2)  41(2)
 37   36(2)                                              38(2)  41(9)
 38   37(2)                                              39(2)
 39   38(2)                                              40(2)
 40   39(2)   6(6)
 41   36(2)  37(9)                                       42(2)  45(2)
 42   41(2)                                              43(2)  45(9)
 43   42(2)                                              44(2)
 44   43(2)   9(6)
 45   41(2)  42(9)                                       46(2)
 46   45(2)  11(6)
 47   29(3)                                              24(1)
 48   29(3)  63(3)
 49   29(3)                                              24(1)
 50                                                      51(2)
 51   50(2)                                              52(2)
 52   51(2)                                              29(4)
```

```
53  17(2)                                                   30(4)
54  18(2)                                                   31(4)
55  19(2)                                                   32(4)
56                                                          57(2)
57  56(2)   11(7)                                           55(2)
58  57(2)                                                   53(4)
59  33(3)   78(3)                                           24(1)
60  70(2)                                                   54(4)   74(4)
61  34(3)                                                   26(1)
62  34(3)                                                   64(8)
63  67(4)                                                   66(3)   46(3)   69(3)   85(3)
64  13(2)   62(8)                                           35(4)
65                                                          67(2)
66  35(3)                                                   67(3)
67  65(2)   66(8)                                           63(4)
68  63(3)                                                   23(1)
69  63(3)                                                   27(1)
70                                                          60(2)
71
72
73
74  75(4)   60(4)                                           65(3)
75  12(2)                                                   74(4)
76  77(4)                                                   65(3)
77  14(2)                                                   76(4)
78  79(4)                                                   59(3)   65(3)
79  15(2)                                                   76(4)
80  81(4)                                                   65(3)
81  16(2)                                                   80(4)
82  29(3)
83  33(3)
84  34(3)
85  35(3)   63(3)   74(3)   76(3)   78(3)   80(3)
```

Figure 10.5. Precedence Matrix Report for the MODEL Specification in Figure 4.10.

10.5 Syntax Error Report.

This error report is generated during the syntax analysis phase and contains the syntax errors during the syntax analysis (Phase -1). No syntax errors were detected for the RUN-1 specification in Figure 4.5. Therefore, to illustrate this error report, consider the set of statements given in Figure 10.6. Four syntax errors were detected for this example, and are shown in the error listing in Figure 10.7.

Note that whenever a syntax error is detected, the complete statement in which the error was detected is discarded. The end of statement delimitor ';' is used for this purpose. An an example, the second error in Figure 10.5, has discarded the statement (line 40) till the semicolon. However, the third error (in Figure 10.5), has deleted the whole line (line 50), because it could not find any semicolon in that line.

The syntax error report is output to the file SAPERR, and the error report is generated in the routine $POPF.

```
                          SOURCE LISTING AND GENERATED STATEMENTS

STATEMENT
NUMBER
   1      A IS GROUP;                                                                     00000010
   2        X IS FIELD(A);.                                                               00000020
   3      B IS GROUP;                                                                     00000030
   4        Y IS FIELD(B,(10);   Z IS FIELD(B);                                           00000040
   5        YY IS FIELD(B,(10) + Z IS FIELD(B)                                            00000050
   5        IF A = B THEN C = REDUCE X + Y;                                               00000060
   6                                                                                      00000061
   6                                                                                      00000062
   6                                                                                      00000063
   6                                                                                      00000064
   5      /*************************************************************************/ ********
   5      /*                                                                       */ ********
   5      /*   FOLLOWING STATEMENTS WERE GENERATED BY THE SYSTEM                   */ ********
   5      /*                                                                       */ ********
   5      /*************************************************************************/ ********
   6      C IS INTERIM;                                                                   ********
   7      REDUCE IS INTERIM;                                                              ********
   8      $COUTREC IS RECORD;                                                             ********
   9      $CINREC IS RECORD;                                                              ********
  10      SYSPRINT IS FILE;                                                               ********
  11      SYSIN IS FILE;                                                                  ********
  12      $DDISK IS MEDIA;                                                                ********
  13      $DDISK IS MEDIA;                                                                ********
```

Figure 10.6. A Simple MODEL Specification.

```
      40     Y IS FIELD(B•(10);  Z IS FIELD(B);
                              |
*VAR11 ERROR.  INVALID TEXT BEGINNING ';' IN LINE NUMBER       40
               REST OF THE LINE TILL THE SEMICOLON IS DISCARDED.

      50     YY IS FIELD(B•(10) + Z IS FIELD(B)
                                |
*VAR11 ERROR.  INVALID TEXT BEGINNING '+' IN LINE NUMBER       50
               REST OF THE LINE IS DISCARDED.

      60     IF A = B THEN C = REDUCE X + Y;
                                         |
*EQUAL ERROR.  INVALID TEXT BEGINNING ';' IN LINE NUMBER       60
               REST OF THE LINE TILL THE SEMICOLON IS DISCARDED.
```

Figure 10.7. Syntax Error Report for the MODEL Specification in Figure 10.5.

10.6 Semantic Error Report.

This report is generated in Phase-3 of the MODEL Processor. This report contains errors due to inconsistency and circularity in the MODEL specification. Again, in the RUN-1 specification of Figure 4.5, no semantic errors were detectd. Therefore, to illustrate this error report, consider the example shown in Figure 10.8. The assertion in this example is inconsistent for the reason mentioned in the corresponding error report of Figure 10.9. Note that, if the user had not specified the assertion at all, then the system would have generated an entirely different assertion (using the reduction function ANY), which is given by:

```
    OA.C = ANY(A.C);
```

In this case, the subscripts on the two sides of the assertion are consistent.

The semantic error report is output to the file ERROR (circularity errors are output to the file ADJMRPT, see GAN78), and the procedure WERROR is used to output a message to the file ERROR.

```
                    SOURCE LISTING AND GENERATED STATEMENTS

STATEMENT
NUMBER
    1     SOURCE FILES: A;                                                  00000001
    2     TARGET FILES: B;                                                  00000002
    3      A IS FILE;                                                       00000010
    4        B IS RECORD(A,(*));                                            00000020
    5            C IS FIELD(B,(*));                                         00000030
    6      CA IS FILE;                                                      00000040
    7        B IS RECORD(CA);                                               00000050
    8            C IS FIELD(B,(*));                                         00000060
    9      CA.C = A.C;                                                      00000070
   10                                                                       00000080
   10                                                                       00000090
   10                                                                       00000100
   10                                                                       00000110
   10                                                                       00000120
   10                                                                       00000130
   10                 /*  FIGURE 10.8   */                                  00000140
    9     /*****************************************************************/ ********
    9     /*                                                               */ ********
    9     /*   FOLLOWING STATEMENTS WERE GENERATED BY THE SYSTEM           */ ********
    9     /*                                                               */ ********
    9     /*****************************************************************/ ********
   10     EXIST IS GROUP;                                                   ********
   11     A IS GROUP(EXIST);                                                ********
   12     B IS INTERIM(A);                                                  ********
   13     CA IS GROUP(EXIST);                                               ********
   14     B IS GROUP(CA);                                                   ********
   15     C IS INTERIM(B);                                                  ********
   16     C IS INTERIM(B);                                                  ********
   17     $DISK IS MEDIA;                                                   ********
   18     $DISK IS MEDIA;                                                   ********
   19     $B1001 IS SUBSCRIPT((1,*,1,1));                                   ********
   20     $B1002 IS SUBSCRIPT((1,*,1,1));                                   ********
```

FIGURE 10.8. Example-2.

```
ERRORS.
   INCONSISTENCY IN SUBSCRIPT SPECIFICATION IN THE ASSERTION: $A0001. THE SUBSCRIPT ON THE RHS: (FOREACH$B,FOREACH$C) IS NOT
         CONSISTENT WITH THE SUBSCRIPT LIST ON THE LHS: (FOREACH$C)
```

FIGURE 10.9. ERROR REPORT for Example-2 in Figure 10.8.

10.7 Warnings.

Warnings are generated throughout the first four phases of the MODEL Processor. Whenever an assertion or a DD statement is changed (or generated) a warning message is sent. In addition, some form of redundencies in the specification (like, subscript redundency) is also reported as warnings. The warning report for the RUN-1 example in Figure 4.5 is shown in Figure 10.10.

The warning messages are output to the file WARNING. The procedure WWARN is used to output a message to the WARNING file.

```
WARNINGS.
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   33. ASSERTION NAME IS ASSUMED TO BE: "$A0001".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   34. ASSERTION NAME IS ASSUMED TO BE: "$A0002".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   35. ASSERTION NAME IS ASSUMED TO BE: "$A0003".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   36. ASSERTION NAME IS ASSUMED TO BE: "$A0004".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   37. ASSERTION NAME IS ASSUMED TO BE: "$A0005".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   38. ASSERTION NAME IS ASSUMED TO BE: "$A0006".
  NO ASSERTION NAME IS SPECIFIED IN STATEMENT NUMBER   39. ASSERTION NAME IS ASSUMED TO BE: "$A0007".
  THE FOLLOWING ASSERTION IS GENERATED WHILE ANALYZING THE ASSERTION ON STATEMENT   33: "$A$I0001: $I0001=QUANTITY(J) *
        UNIT_PRICE;".
  THE ASSERTION ON STATEMENT   33 IS CHANGED TO THE FOLLOWING: "$A0001: PRINTER.REPORT_1.PAGE.REP_1.PRODUCT_TOTAL_FOR_MONTH=SUM(
        $I0001);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "POINTER".  THE FOLLOWING STATEMENT IS GENERATED: "POINTER IS GROUP;".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "ENDGRP".  THE FOLLOWING STATEMENT IS GENERATED: "ENDGRP IS GROUP;".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "MASTER".  THE FOLLOWING STATEMENT IS GENERATED: "MASTER IS GROUP(TAPE);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "PRODUCT".  THE FOLLOWING STATEMENT IS GENERATED: "PRODUCT IS GROUP(TRANSACTION);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "TRANSACTION".  THE FOLLOWING STATEMENT IS GENERATED: "TRANSACTION IS GROUP(TAPE);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "TAPE".  THE FOLLOWING STATEMENT IS GENERATED: "TAPE IS GROUP(EXIST);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "PAGE".  THE FOLLOWING STATEMENT IS GENERATED: "PAGE IS GROUP(REPORT_1);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "REPORT_1".  THE FOLLOWING STATEMENT IS GENERATED: "REPORT_1 IS GROUP(PRINTER);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "PRINTER".  THE FOLLOWING STATEMENT IS GENERATED: "PRINTER IS GROUP(EXIST);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "EXIST".  THE FOLLOWING STATEMENT IS GENERATED: "EXIST IS GROUP;".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "MAST".  THE FOLLOWING STATEMENT IS GENERATED: "MAST IS INTERIM(POINTER,(*));".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "TRANS".  THE FOLLOWING STATEMENT IS GENERATED: "TRANS IS INTERIM(ENDGRP,(*,*));".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "$I0001".  THE FOLLOWING STATEMENT IS GENERATED: "$I0001 IS INTERIM((*,*));".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "MAST".  THE FOLLOWING STATEMENT IS GENERATED: "MAST IS INTERIM(MASTER);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "TRANS".  THE FOLLOWING STATEMENT IS GENERATED: "TRANS IS INTERIM(PRODUCT);".
  NO DD STATEMENT IS SPECIFIED FOR THE NAME: "REP_1".  THE FOLLOWING STATEMENT IS GENERATED: "REP_1 IS INTERIM(PAGE);".
  THE FOLLOWING STATEMENT IS GENERATED TO RESOLVE INCOMPLETENESS: "FOREACH$MAST IS SUBSCRIPT(MAST,(1,EXIST.TAPE.MASTER.MAST,1,1));"
```

Figure 10.10. Warnings for the MODEL Specification in Figure 4.10.

CHAPTER 11

CONCLUSIONS

This Chapter briefly describes the last two phases of the MODEL Processor, namely, Sequencing and Code Generation Phases. The design and implementation of these two phases are not part of this dissertation, however, the techniques used in the earlier version of the system (PRY76A) can be used in the design of these phases. The Chapter also summarizes the conclusions that can be drawn based on the research described in the dissertation.

11.1 Sequencing and Optimization.

This phase of the MODEL Processor determines the sequence of execution of all events implied by the MODEL specification using conventional Graph Theory techniques. The input to this phase is a cycle free directed graph obtained from the previous phase. If there are cycles, they are reported to the user in the procedure CYCLES described in the previous chapter (and in GAN78).

The sequencing phase also performs the scope and iteration analysis. Therefore, the result of this phase is a set of data structures representing the desired sequence of processes and flow of events, sequenced and ranked in their order of execution. The recursions involved must also be scheduled, using the procedure SCHEDUL described in the previous chapter.

There is another function performed by the sequencing phase. That is, the optimization of the sequence and the data structures in the sequence. The optimization techniques can be categorized into three types: (1) Memory optimization, (2) Execution time optimization and (3) I/O Channel time optimization. These are briefly described in the following.

11.1.1 Memory Optimization.

This generally means generating optimized internal storage structure that reduces memory requirements. For example, for the following set of statements:

```
F(1) = 1;
FOR N = 2 TO 100 F(N) = F(N-1)*N;
B = F(10);
C = F(100);
```

the completeness analysis process would have generated a data description statement for F, which is given by:

```
F IS INTERIM((100));
```

and the directed graph for the above set of assertions will be as shown in Figure 11.1. For this graph, we can obtain the following schedule (using PL/1 notation):

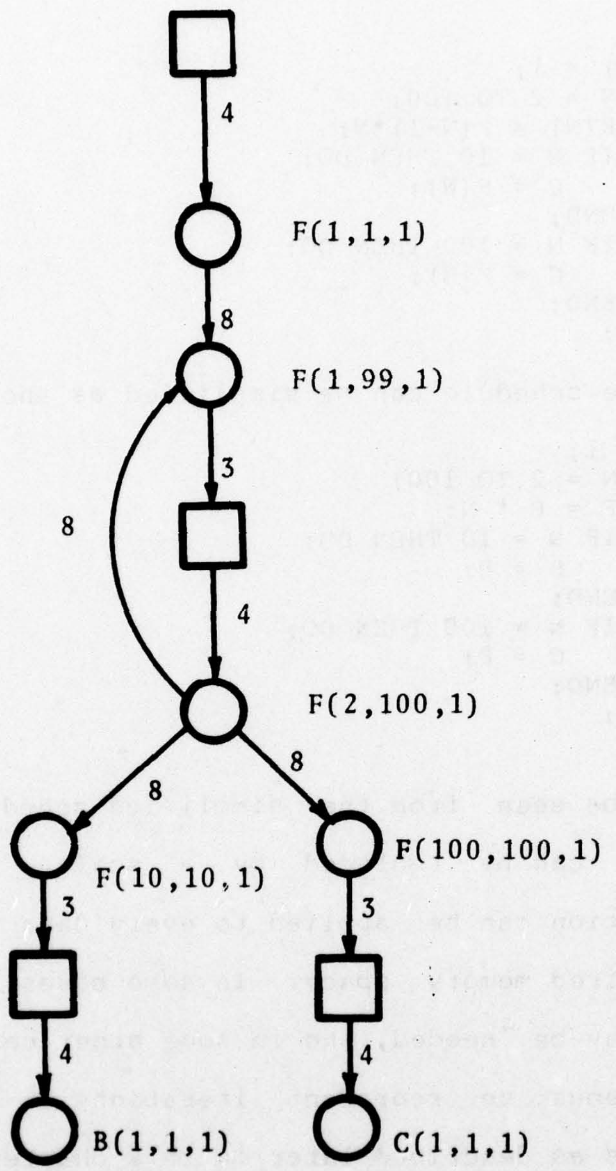


Figure 11.1. Directed graph for the Example in Section 11.1.1.

```
F(1) = 1;
DO N = 2 TO 100;
   F(N) = F(N-1)*N;
   IF N = 10 THEN DO;
      B = F(N);
   END;
   IF N = 100 THEN DO;
      C = F(N);
   END;
END;
```

The above schedule can be simplified as shown below:

```
F = 1;
DO N = 2 TO 100;
   F = F * N;
   IF N = 10 THEN DO;
      B = F;
   END;
   IF N = 100 THEN DO;
      C = F;
   END;
END;
```

As can be seen from the simplified schedule above, the array F can be replaced by a scalar. Such type of optimization can be applied to every data name to reduce the required memory space. In some cases, more than one buffer may be needed, and in some other cases, it may be advantageous to represent iterations in the graph by functions as described later in this chapter.

Similar optimization technique can be applied to the input and output data structures. For example, if a file is defined as consisting of 100 records by the following statement:

```
INREC IS RECORD(INPUT_FILE,(100));
```

then it is not necessary to represent the record as an array of 100 elements. Most of the times, it is enough to represent only one instance of such set of records by allocating space for only the first record and overlaying the other records on the first. However, this is not possible if the contents of more than one records are used as source of any assertion. Consider, for example, the set of MODEL statements shown in Figure 11.2. For these set of statements, only the I th and I+1 th records (in the array RECA) are needed at any instant of time. Therefore, if the structure of the group of records in the INPUT file is declared by the following PL/1 declarations:

```
DCL INPUT FILE RECORD INPUT;
DCL 1 GA BASED($PGA),
      2 RECA,
        3 CUST# PIC '99',
        3 AMOUNT PIC '999V99';
```

```
INPUT IS FILE;
  GA IS GROUP(INPUT,(*));
    RECA IS RECORD(GA,(*));
      CUST# IS FIELD(RECA,PIC'99');
      AMOUNT IS FIELD(RECA,PIC'999V99');

OUTPUT IS FILE;
  RECB IS RECORD(OUTPUT,(*));
    CUST# IS FIELD(RECB,PIC'99');
    AMOUNT IS FIELD(RECB,PIC'999V99');

RECB.AMOUNT = SUM(RECA.AMOUNT);

IF RECA.CUST# (I) = RECA.CUST#(I+1)
    THEN ENDGRP.RECA = TRUE;
    ELSE ENDGRP.RECA = FALSE;
```

Figure 11.2 A Simple MODEL Specification.

and $NX_PGA is a pointer to the next record (I+1 th record), then the current record and the next record at any instant can be accessed using the two pointers $PGA and $NX_PGA respectively. In this case, the last assertion in Figure 11.2 must be denoted by the following PL/1 statement:

```
    IF RECA.CUST# = $NX_PGA -> RECA.CUST#
       THEN ENDGRP.RECA = TRUE;
       ELSE ENDGRP.RECA = FALSE;
```

This technique has been used (for restricted cases) in MODEL-II (PRY76A).

11.1.2 Execution Time Optimization.

There are many ways of optimizing a program to reduce execution time. For example, transfer of an expression from a loop will sometimes reduce the execution time significantly. For example, in the expression:

```
     B(*) = C(*) * SQRT(N);
```

the function SQRT is executed for each index value. If the above expression is changed to:

```
     TEMP = SQRT(N);
     B(*) = C(*) * TEMP;
```

the function SQRT is executed only once, and therefore, can save significant amount of execution time. The standard optimization techniques used in compilers like, PL/1 Optimizer can be used.

11.1.3 <u>I/O Channel Time Optimization</u>.

I/O channel time overhead of any program is dependent on the type of file organization. For example, the user might have fixed record format for a file. By changing the record format to fixed blocked type, the number of EXCPs required to perform the I/O operations can be reduced significantly.

11.2 <u>Code Generation</u>.

The function of this phase of the Processor is to generate the program in the target language (PL/I) from the optimized flowchart obtained from the previous phase. Because, MODEL is a higher language than the target language PL/I, the generation of code from the flowchart is not straight forward. Some of the additional processing required during the code generation are summarized in the following:

(1). Structuring of the files: Sometimes it is necessary to restructure a file specified in MODEL to an acceptable structure in the target language. This restiucturinq may involve packinq (in case of target files) or unpacking (in case of source files) of the records in a file.

(2). Handling abonormal terminations (in the generated program) like, error conditions, end of file conditions, and so on.

(3). Handling special keywords like, ENDGRP, SUBSET, ABSENT, and so on, used in the MODEL specification.

Techniques used by RIN (RIN76) and Pnueli (PRY76A) can be extended to perform the above functions of the Processor. Some of the required techniques are briefly described in the following.

11.2.1 <u>Use of Functions</u>.

Usually, each assertion can be translated into a set of target language statements in appropriate sequence. However, in some cases, a set of assertions may have to be represented by a function or a procedure. This function can even be recursive, if the assertions under consideration define the elements of a data name recursively. For example, consider the following set of assertions:

```
F(1) = 1;
F(N) = F(N-1)*N;
A = F(X);
B = F(Y);
```

In this case, because X and Y are unknown, it is more appropriate to treat F as a recursive function, and not as an array. (Note that, as mentioned in the earlier section, F can be represented by a scalar (or an array) if X and Y are both constants.) Generation of code for recursive procedures from the directed graph can be quite easy in some target languages like PL/1, but some additional effort may be required while using some other target languages (like COBOL).

11.2.2 Packing and Unpacking Records.

Packing and unpacking of records, described in the beginning of this section, requires further discussion. If a record contains a data name that repeats variable number of times (or the length of that data name varies), then in both MODEL-I and MODEL-II, it is required that the maximum allowed dimension (or the length) for that data name is specified by the user. This maximum value is used in allocating the record, and eventually, the individual characters in the actual record are unpacked to the allocated record (in the case of a source file record; in the case of a target file record, the allocated record is packed to obtain the required record (RIN76)). This packing (and unpacking) is illustrated in the examples of

```
A IS RECORD;
  N IS FIELD(A,PIC'99');
  X IS FIELD(A,(0:100),CHAR(40));
  Y IS FIELD(A,PIC'999');

EXIST.X = N;
```

Figure 11.3. An example that requires Unpacking.

```
DCL BUF CHAR(4006) VAR;
DCL 1 A,
        2 N PIC '99',
        2 X(100) CHAR(40),
        2 Y PIC '9999';

READ FILE(FILENAME) INTO(BUF);

I = 1;
N = SUBSTR(BUF,I,2);
I = I + 2;
EXIST.X = N;
DO J = 1 TO EXIST.X;
   X(J) = SUBSTR(BUF,I,40);
   I = I + 40;
END;
Y = SUBSTR(BUF,I,4);
```

Figure 11.4. Unpacking an Input Record.

Figures 11.3 and 11.4. In Figure 11.3, the record A is shown to contain a field, X, which repeats variable number of times (at most, 100). And, Figure 11.4 shows the generated declaration for the record and the unpacking statements that are used to move the actual record (i.e., BUF) into the declared record, A.

It is not necessary to perform the type of unpacking (or packing), shown in Figure 11.4, in all cases. For example, there is no need of unpacking for the specification of Figure 11.3, if the record is decalred as shown in Figure 11.5.

```
DCL 1 A,
        2 N PIC '99',
        2 X (NN REFER(N)) CHAR(40),
        2 Y PIC '999';
```

Figure 11.5. Using REFER option.

In the declaration of Figure 11.5, the REFER option of PL/1 is used to define the size of the repeating variable, X. This technique can be used in most cases. However, even in PL/1 optimizer, the use of refer option is restricted, and therefore, cannot be used in the following cases:

(1). When the variable that defines the repetition (RHS argument of the refer option) is not in the structure as shown in the example of Figure 11.6.

(2). When the variable that defines the repetition is in the same structure, but physically located after the repeating variable as in the example of Figure 11.7.

(3). When the variable that defines the repetition is an array as in the example of Figure 11.8.

```
A IS RECORD;
  B IS FIELD(A,(N),CHAR(10));
  C IS FIELD(A,CHAR(10));
```

Figure 11.6.

```
A IS RECORD;
  B IS FIELD(A,(N),CHAR(10));
  N IS FIELD(A,PIC'99');
```

Figure 11.7.

```
A IS RECORD;
  B IS GROUP(A,(*));
    C IS FIELD(B,PIC'99');
    D IS FIELD(B,(*),CHAR(10));

EXIST.D(I) = C(I);
```

Figure 11.8.

```
DCL 1 A BASED($AP),
      2 $N PIC '99',
      2 B (XN REFER($N)) CHAR(10),
      2 C CHAR(10);

DCL BUF CHAR(MAXSIZE) VAR BASED(P);
DCL BUF1 CHAR(MAXSIZE) VAR BASED($AP);

ALLOCATE BUF;
UNSPEC($AP) = UNSPEC(P) + 2;

/*NOW READ THE FILE*/

READ FILE(FILENAME) INTO(BUF1);
$N = N;
   ...
```

Figure 11.9. PL/1 code for the example in Figure 11.6.

```
DCL 1 A BASED($AP),
      2 $N PIC '99',
      2 B (XN REFER($N)) CHAR(10),
      2 N PIC '99';

DCL BUF CHAR(MAXSIZE) VAR BASED(P);
DCL BUF1(MAXSIZE) VAR BASED($AP);

ALLOCATE BUF;

UNSPEC($AP) = UNSPEC(P) + 2;

/*READ THE FILE NOW*/

READ FILE(FILENAME) INTO(BUF1);

$N = SUBSTR(BUF1,LENGTH(BUF1)-1,2);
```

Figure 11.10. PL/1 code for the example in Figure 11.7.

In the first two cases, the file structure can be expanded slightly by including the data names that define the dimensions (of the repeating data names) in the beginning of the declared PL/1 structure. Then, the refer option can be used as usual. For example, the PL/1 codes for reading the records in the examples of Figures 11.6 and 11.7 are shown in Figures 11.9 and 11.10 respectively.

However, in the third case, it is probably easier to use the unpacking and packing technique as done in MODEL-I and MODEL-II.

If the target language does not provide such facilities as the refer option mentioned above, then packing and unpacking of data is necessary. Note that the amount of restrictions imposed by PL/1-F (if used as a target language) on the use of refer option is more than that of PL/1 Optimizer.

11.3 <u>Conclusions</u>.

In the previous chapters, the important features of the MODEL Processor were described. MODEL is described as a simple non-procedural language in which a non-programmer can specify his data processing requirements easily. The language has the power of very high level languages like APL, where array operations can be expressed without using subscripts. The importance of such ability is not in expressing them, but in analyzing such operations for completeness and consistency. The array graph representation described in Chapter 9, has served as an excellent tool for the analysis of iterations and recursions in the MODEL specification.

11.3.1 Limitations of the Implementation.

Even though some criteria for splitting assertions were described in Chapter 9, they were not implemented in the MODEL Processor. This is because of lack of efficient algorithms to perform those splittings (or decompositions). Besides, we were not sure if we want to make the system too powerful, while advocating its use for untrained users.

Also, we have restricted ourselves to the analysis of recursions on only one dimension. Consistency analysis of recursive specifications on more than one dimensional arrays has not been tackled fully, and is a topic for further research.

Even in single dimension case, we have restricted ourselves to linear subscript expressions. Only with this restriction, could we arrive at the conditions for consistency and sequenceability (described in Chapter 9).

Analysis of a general case is quite complicated. Even in the simple linear case, we have restricted the range to a finite domain (rather than infinite domain, which was suggested by Dr. Pnueli). Extension of the results of finite domain to infinite domain may also be quite interesting.

It was our experience that an untrained user has the most difficulty in specifying proper subscripts for iterative specifications of a problem. Therefore, it was quite important for us to accept subscript-less specification. If the subscripts are not specified, they are automatically supplied, and if no range for a subscript is specified, then an appropriate range (usually from the data description statement of the data name with which the subscript is associated) is supplied. However, care must be used in specifying complex operations like, outer product. For example, if B is a matrix, and A is a vector, then there is no way of expressing the relation:

$$\forall\, i,j \qquad B(I,J) = A(I) * A(J)$$

using our subscript-less language (at least in the current implementation)*. Of course, this type of problem is resolved in higher languages, like APL, by using special operators.

11.3.2 Interpretation of Assertions.

Assertion splitting is not really necessary if we are willing to "interpret" the assertions in the loop which is found to be unschedulable. This may lead to inefficient programs and reporting of some of the inconsistencies may have to be delayed until the execution of the generated program. For example, for the set of assertions given in Figure 9.14, we can obtain the PL/1 program shown in Figure 11.11.

---

* If the assertion:

```
B = A * A;
```

is specified in MODEL, where B is a matrix and A is a vector, then the assertion is changed to the following:

```
B(I,J) = A(J) * A(J)
```

```
DCL $$A(100) BIT(1) INIT((100) '0'B);
DCL $ASW BIT(1) INIT('1'B);


   DO WHILE ($ASW);
      $ASW = '0'b;
A1:   /* ASSERTION: A1  */
        IF $$A(1) THEN DO;
             A(1) = 0;
             $ASW = '1'B;
             $$A(1) = '1'B;
        END;


A2:   /* ASSERTION: A2  */
        DO I = 1 TO 50;
             IF $$A(2*I) THEN
             /* IF A(2*I) NOT COMPUTED*/
               IF $$A(I) THEN DO;
               /* AND A(I) IS COMPUTED*/
                  A(2*I) = A(I)+1;
                  $ASW = '1'B;
                  $$A(2*I) = '1'B;
             END;
        END; /*DO I*/
```

Figure 11.11.

```
A3:    /* ASSERTION: A3  */
       DO I = 31 TO 50;
            IF $$A(2*I-1) THEN
            /* IF A(2*I-1) NOT COMPUTED*/
              IF $$A(2*I) THEN DO;
              /* AND A(2*I) IS COMPUTED*/
                 A(2*I-1) = A(2*I)+2;
                 $ASW = '1'B;
                 $$A(2*I-1) = '1'B;
            END;
       END; /*DO I*/

A3:    /* ASSERTION: A4  */
       DO I = 1 TO 29;
            IF $$A(2*I+1) THEN
            /* IF A(2*I+1) NOT COMPUTED*/
              IF $$A(2*I+5) THEN DO;
              /* AND A(2*I+5) IS COMPUTED*/
                 A(2*I+4) = A(2*I+5)+1;
                 $ASW = '1'B;
                 $$A(2*I+1) = '1'B;
            END;
       END; /*DO I*/
   END; /*END DO WHILE*/
```

Figure 11.11.

```
/*NOW CHECK FOR COMPLETENESS AND CONSISTENCY*/
IF ANY($$A) THEN DO;
     /*SOME ELEMENTS ARE NOT COMPUTABLE
       SEND AN ERROR MESSAGE*/
END;
•
•
```

Figure 11.11. A PL/1 Program for the Assertions in Figure 9.14.

In the Program of Figure 11.11, an array of switches ($$A) is associated with the array A. An element of $$A is set if the corresponding element in A is computed. The applicability of each assertion to compute an element is tested repeatedly till no more computation is possible ($ASW = '0'), or till all the elements are computed.

The interpretation technique described above is quite useful because it can be applied not only to unschedulable loops, but also to the loops containing nodes whose bounds are non-integers. However, as can be seen from the program of Figure 11.11, it may generate inefficient programs. In some cases, a better solution can be obtained by moving part of interpretation (during the execution phase) to the consistency analysis phase. Consider, again, the set of assertions in Figure 9.14. We can move all the interpretations (implied by the program in Figure 11.11) to the analysis phase, and also, during the interpretation we can obtain an ordered set of elements that were computed using each assertion, in each pass in the "DO WHILE" loop. For the example under consideration, we can see that during the first pass we can compute the elements:

1	using A1,
2,4,8,16,32,64	using A2,

63 using A3 and

59,55,51,47,43,39,35,31,27,
23,19,15,11,7,3 using A4.

During the second pass, we can compute the elements:

6,12,14,22,24,28,30,38,44,46,
48,54,56,60,62,70,76,78,86,
88,92,94,96 using A2,

61,69,75,77,85,87,91,93,95 using A3 and

57,53,49,45,41,37,33,29,
25,21,17,13,9,5 using A4.

And, finally, during the third pass, we can compute the elements:

10,18,20,26,34,36,40,42,50,52,
58,66,68,72,74,80,82,84,90,98,
100 using A2 and

65,67,71,73,79,81,83,89,97,99 using A3.

From the above set of numbers, we can generate (even though, it is not trivial) the required PL/1 program, part of which is shown in Figure 11.12.

```
/* PASS 1*/

     /*ASSERTION: A1*/

     A(1) = 0;

     /*ASSERTION: A2*/

     I = 1;

     DO WHILE (I<50);

          A(2*I) = A(I) +1;

          I = 2*I;

     END;

     /*ASSERTION: A3*/

     A(63) = A(64) + 2;

     /*ASSERTION: A4*/

     DO I = 29 TO 1 BY -2;

          A(2*I+1) = A(2*I+5)+4;

     END;

/* PASS 2*/

     /*ASSERTION: A2*/

     DO I = 3 TO 47 BY 4;

          A(2*I) = A(I) + 1;

     END;

     DO I = 6 TO 46 BY 8;

          A(2*I) = A(I) + 1;
```

Figure 11.12.

```
END;
DO I = 12 TO 44 BY 16;
     A(2*I) = A(I) + 1;
END;
A(48) = A(24) + 1;
A(96) = A(48) + 1;

/*ASSERTION: A3*/

    .

    .

    .
```

Figure 11.12. Part of PL/1 program for the assertions in Figure 9.14.

Note that the program in Figure 11.12 is longer than that in Figure 11.11 because of non-linear nature of the computations. However, the program in Figure 11.12 does not use any additional array of switches (like $$A in Figure 11.11). To obtain the program in Figure 11.12, we need a set of algorithms to split a set of elements obtained in any pass into subsets of elements that can be represented by a regular do loop. Development of such algorithms, or better techniques than the ones described above needs further research. Note that the program generated for the graph is Figure 9.14 from the assertion splitting procedures described in Chapter 9 will be almost similar to the program shown in Figure 11.12.

11.3.3 Comments on Man-machine Interface.

There is one additional drawback with the current implementation. Except during syntax analysis phase, a detected error is reported to the user, but no dialogue between the user and the processor is allowed. In other words, the user cannot respond to any of the error reports. During the syntax analysis, however, an error is reported to the user as soon as it is detected, and the user can immediately reenter the line, thus rectifying the error. Even such a dialogue is not too good, and a "true" automatic programming system should have the facility to invoke an editor. Provisions must also be provided to process user responses in the later phases of the MODEL processor, otherwise, debugging a specification can be quite laborious and inefficient.

Despite the restrictions mentioned above, the analysis techniques described in this dissertation have greatly reduced the effort required by the user for specifying a problem, and is a big step in the automation of software development process. Even though much research in this field lies ahead (as can be seen from the above observations), we feel that this research has made a big contribution towards achieving a simpler man-machine interface.

APPENDIX A

In this appendix, a brief description of each subroutine used in MODEL Processor is given. The first column is the procedure name or entry point name. Second column indicates the procedure name in which the entry name in column 1 is located. Third column indicates the arguments of the procedure or entry point. Standard conventions are used to indicate the attributes of the arguments. Each argument is prefixed by a number or '*'. I th argument is prefixed by the number "i". If the procedure is used as a function (using the returns attribute), then the return attributes are prefixed by '*'. Column 4 indicates all the routines that the procedure calls. Note that this column is empty if the first column represents an entry point. Fifth column indicates the routines that use the procedure or the entry point. And, finally, the last column gives a brief description of the functions of the procedure or entry point.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SKL-FRF	SHOPF			SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
SFAIL	SHOPF			SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
SNAFU	SHOPF			SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
SHOPF			LEXGIS LEXGI LEXFAIL SPECLEUF	SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
SPUSH	SHOPF	1-(*)CHAR(6)		SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
SSUCCES	SHOPF			SAP	"HOUSE KEEPING" ROUTINE FOR SAP.
ACRADJ			ACRDR ADJNSET GETFNAM ACRGRHP SETTSG	MONITOR	CREATES THE PRECEDENCE MATRIX.
ACRCHST			ACRDB ACRDT ACRGGNP GENSTMT RETRIVE RETNFE RETRNAM SETDTHP TESTFUN	ACRSTVS	CHECKS FOR THE EXISTENCE OF DD STATEMENTS FOR THE SOURCE AND TARGET VARS OF AN ASSERTION.
ACRDDF	ACRDUP	1-POINTER		DUPLUDE	CREATES DUPLICATE DICTIONARY ENTRIES FOR THE FIELDS IN AN UPDATE FILE.
ACRDFF	ACRDC	*-POINTER		PFTDICT RESIC SETBOFE SUPFORE	RETURNS THE POINTER TO THE FIRST DICTIONARY ENTRY STACK.
ACRDICT			ACRDUP ACRPTR ACRSTVS ASUBST GENSTMT IMEDIA IUSERG RETREVE RETNPE RETRNAM RETHPRX SCOMSET STORE UPDDTR WWARN1 WWARN2 WSAPOH LINKSEQ PSORT DUPLUDE	MONITOR	CREATES THE DICTIONARY OF DATA NAMES.
ACRDNP	ACRDC	1-POINTER		SETSUBN	RETURNS IN THE SECOND ARGUMENT, ANOTHER DICTIONARY ENTRY FOR THE DICTIONARY INDEX GIVEN BY THE FIRST ARGUMENT.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
ACRPDA	ACRPD	1-BIN FIXED 2-POINTER		ACRADJM CHKSCON CSUFLST CSUBRED CYCLES DUMPVL DUPLDSK GANMFTV GDINDEX GDTAL GREDFUN IFENAMS MERGSUB NODTYPE PRTGNM RANGSET RESICR RICOMPE RICONSE SETPR SETSURN SETTS SORTASS SUBPRC SUBPROP SUPFORE TESTMED UPDASSN UPVLIST XWXREF	RETURNS THE STORAGE ENTRY POINTER OF THE DATA NAME NAME REPRESENTED BY THE FIRST ARGUMENT INDEX.
[illegible]	ACRPD	1-BIN FIXED 2-POINTER		ADJMRP ADJMRS ADJMRP CYCLES DUPLDSK GANMFTV GCNODEL IFENAMS OPT#P REMOVES RESIC SORTASS SUBPROP UPDASSN	RETURNS THE POINTER TO THE PRECEDENCE MATRIX ENTRY LIST FOR A PARTICULAR ROW OR A COLUMN.
[illegible]	ACRPD	1-BIN FIXED 2-POINTER		ADJMRS	SAVES THE POINTER TO THE PRECEDENCE MATRIX ENTRY LIST OF A ROW OR COLUMN.
[illegible]	ACRPD	1-BIN FIXED 2-(*,*)BIN FIXED		UPDDIM UPVLIST	UPDATES THE POINTER TO THE VARIABLE LIST OF A DICTIANARY ENTRY.
ACRDIR		1-POINTER	ACRDC ACRGGNM ERSYST GETGNM ACRDUP2 RETREVE RETRPL RETRNAM SETDTRP	ACRDICT	UPDATES THE DICTIONARY TO INCLUDE THE STATEMENT REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
[illegible]		1-PTR		ACRDUP DUPLSE	UPDATES STATUS BITS IN THE STRUCTURE DATA-LIST OF THE STORAGE ENTRY REPRESENTED BY THE ARGUMENT POINTER.
ACRDUP		1-POINTER	NULLADJ GINDEX2 GSBITS RVINDT	ACRCHST ACRDUP ACRPTR ASUBST DUPLSE GENASSI GENSTMT IFENAMS RESIC SETDTRP UPDICT	CREATES A DICTIONARY ENTRY FOR THE DATA NAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
[illegible]	ACRPD	1-POINTER 2-BIN FIXED 3-BIN FIXED		ACRCHST RICOMPE	CREATES A DICTIONARY ENTRY FOR A DATA NAME IN AN ASSERTION.
ACRGGNM	GETGNM	1-POINTER 2-CHAR(100)VAR		ACRADJM ACRCHST ACRDUP ACRPTR	RETURNS THE FULLY QUALIFIED NAME OF A STORAGE ENTRY (INCLUDING THE PREFIX).

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION

ACRMF	ACRCHST	1-CHAR(*)
		2-CHAR(*)
		3-CHAR(*)
		4-POINTER
		5-BIT(1)
				GETFNAM GSTEPTR	USED TO COMPARE THE IDENTITY OF TWO NAMES (FIRST TWO ARGUMENTS).

ACRPOS	ACRDC	1-BIN FIXED		PRTDICT	OBTAINS THE "APPROPRIATE" DICTIONARY STACK FOR A DICTIONARY ENTRY.
ACRPOSP	ACRDC	1-BIN FIXED 2-POINTER		CHKCUNT CSUBL DUPLUDE GETVARS GMAPV GRNGDIM IFENAPS PRTADJL RSORT SETTYPE SUBPROP UPDASSN XWXREF	OBTAINS THE "APPROPRIATE" DICTIONARY STACK FOR A DICTIONARY ENTRY.
ACRPOS1	ACRDC	1-BIN FIXED			OBTAINS THE "APPROPRIATE" DICTIONARY STACK FOR A DICTIONARY ENTRY.
ACRFTA		1-POINTER	ACRD3 ACRCONP GENI GENSTMT GETGNM1 GUVPTR SRCHFIL	ACRDICT	CREATES DICTIONARY ENTRIES FOR THE EXIST LENGTH AND POINTER NAMES.
ACRSTVS		1-POINTER	ACRCHST ASPSPTR GET_AE SETSUBR REPUSUB	ACRDICT GENASSI	THE MAIN ROUTINE THAT SCANS THE DERIVATION TREE OF AN ASSERTION.
ADJP	ADJPPS	1-BIN FIXED 2-BIN FIXED 3-BIN FIXED		GCNODEL RESIC SETT9	RETURNS A PARTICULAR PRECEDENCE MATRIX ENTRY. THE TWO ARGUMENTS SPECIFY THE ROW AND COLUMNS OF THE MATRIX.
ADJPNF		1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-BIN FIXED	ACRDRA PRTHEXC		RETURNS THE # OF SOURCE VARS. IN AN ASSERTION
ADJPPS	ADJPPF	1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-BIN FIXED			RETURNS THE NUMBER OF "P2" SUCCESSORS OF "P1" SUCCESSORS OF DICTIONARY ENTRY "I", WHERE, I, P1, AND P2 ARE THE THREE ARGUMENTS.
ADJPCS	ADJPPS	1-BIN FIXED 2-(*,*)BIN FIXED			SETS AN ARRAY OF ENTRIES IN A COLUMN OF THE PRECEDENCE MATRIX.
ADJPCP	ADJPPF	1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-(*)BIN FIXED			OBTAINS ALL THE SOURCE VARS OF AN ASSERTION.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
ADJMES	ADJMAP	1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-(*)BIN FIXED			OBTAINS ALL TARGET VARS. OF AN ASSERTION
ADJMPS		1-BIN FIXED 2-(*,*)BIN FIXED	ACRDPA ACRDRS DUMPHEX ERSYST		UPDATES A ROW OF THE PRECEDENCE MATRIX.
[illegible]		1-BIN FIXED 2-BIN FIXED	ACRDRA	DUPLDSK OBTAP REMOVED	REMOVES A PARTICULAR ENTRY IN THE PRECEDENCE MATRIX.
ADJMSET	ADJMPS	1-BIN FIXED 2-BIN FIXED 3-BIN FIXED		ACNADJM CHKCONT DUPLDSK GENASSI IFENAMS RESIC RICOMPE SETPR SETTYPE SETT56 SETT9 UPDICT	SETS A PARTICUALR ENTRY IN THE PRECEDENCE MATRIX. FIRST TWO ARGUMENTS SPECIFY ROW AND COLUMN RESPECTIVELY, AND THE LAST ARGUMENT SPECIFIES THE VALLE.
ANALIND		1-POINTER 2-CHAR(1)	GENASSI SUBSCRT REDFUN	GET_AE	ANALYZES A VARIABLE IN AN ASSERTION AND THE ASSOCIATED SUBSCRIPT.
[illegible]	ANALIND	1-POINTER		ACRSTVS	OBTAINS A LIST OF SOURCE AND TARGET VARS FOR AN ASSERTION.
[illegible]		1-PTR 2-BIN FIXED 3-CHAR(*)		DUMPHEX	RETURNS IN THE THIRD ARGUMENT, THE CHARACTER REPRESENTATION OF THE CONTENTS OF THE MEMORY REPRESENTED BY THE FIRST ARGUMENT POINTER.
ASUBST		1-POINTER	ACRDC CSUBLST DATAYPE GETDNM GETRDNM CSTLPTR RETREVE RLTRFE SEMERR SETDTRP UPDDIM	ACRDICT	ANALIZES THE SUBSCRIPT STATEMENT REPRESENTED BY THE ARGUMENT POINTER.
[illegible]	JUNK				
BUILDD			CONVPTR RECLEUF	MONITOR	OBTAINS A DICTIONARY FOR XREF REPORT.
CENTERL	PRTHDR	1-CHAR(*)VAR		CYCLES PRTDDS PRTHDRS	CENTERS THE ARGUMENT STRING IN THE FORMATTED OUTPUT.
CENTRALD	PRTHDR	1-CHAR(*)VAR 2-BIN FIXED		PRTDDS	CENTERS THE ARGUMENT STRNG IN THE FORMATTED OUTPUT WITH AN OFFSET (GIVEN BY THE SECOND ARGUMENT).
CHKCONT		1-BIN FIXED 2-BIN FIXED	ACRPOSP ADJMSET DIGPH ERSYST WEIGHTV	SETTYPE	ESTABLISHES TYPE-8 RELATIONSHIP BETWEEN THE TWO ARGUMENT NODES.

AD-A063 900 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
VERIFICATION AND CORRECTION OF NON-PROCEDURAL SPECIFICATIONS IN--ETC(U)
SEP 78 S K SHASTRY N00014-76-C-0416
UNCLASSIFIED 78-01 NL

6 OF 6
AD
A063900

END
DATE
FILMED
4-79
DDC

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
CHKSCON			CLINC CSMAPV CTMRES GCYCLES ACPOR CONVR DATAARE LISTGET LISTSL LISTSA WEPROR	MONITOR	CHECKS SUBSCRIPT CONSISTENCY.
CLINC				CHKSCON	
CONVT		1-BIN FIXED 2-CHAR(30)VAR		CHKSCON IMEDIA ISPUNIT PALTSNM PRTASSX PRTSTMT WICONSL SUBPROP WWARNP WWARN1 WXREF XWXREF	RETURNS THE PICTURE REPRESENTATION OF THE ARGUMENT NUMBER.
CONVTB	CONVT	1-BIN FIXED(31) 2-CHAR(30)VAR		WXREF XWSIZE	RETURNS THE PICTURE REPRESENTATION OF THE ARGUMENT NUMBER.
CONVTA	CONVT	1-CHAR(*) 2-BIN FIXED 3-CHAR(30)VAR		IMEDIA	PERFORMS SOME SPECIAL FORMATTING OF THE ARGUMENTS.
CONVTV	CONVT	1-BIN FIXED 2-CHAR(30)VAR		IMEDIA IUNIT	PERFORMS SOME SPECIAL FORMATTING OF THE ARGUMENTS.

CONVTZ	CONVT	1-CHAR(*)
		2-CHAR(*)
		3-CHAR(*)
		4-(*)CHAR(*)
		5-CHAR(30)VAR
				IUNIT	PERFORMS SOME SPECIAL FORMATTING OF THE ARGUMENTS.

CONVPTR	ASSNDX	1-POINTER 2-CHAR(8)		BUILDD DUMPHEX GET_AE PRTAM SETMOFE STORE SVANAM TRACE UPDDTRD	RETURNS HEX REPRESENTATION OF ARGUMENT POINTER.
CGRAPV		1-POINTER 2-BIN FIXED	GRAPV LOOPMV	CHKSCON	OBTAINS THE SELF MAP VECTOR ASSOCIATED WITH A CYCLE.
CSUBL		1-POINTER	ACRPOSP STMIDEL	CSUBLST	OBTAINS A LIST OF DATA NAMES THAT USE A SUBSCRIPT NAME.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
CSUBLST			ACRDR CSUBL CSUBRED ERSYST GDINDEX GEXACTD GADFDIM GPPTR REPDTRP RETREVE RETRVE SETDTRP SSUBRNG	ASUBST RESIC	ANALYZES THE LIST OF DATA NAMES ASSOCIATED WITH A SUBSCRIPT NAME.
CSUBRED			ACRDR GPPTR STMTDEL	CSUBLST	ELIMINATES ALL REDUNDENT SUBSCRIPTS.
CTHRES			GADRDV GMAPV SSFACT	CHKSCON	OBTAINS THE SUM OF THRESHOLDS ASSOCIATED WITH A LOOP.
CYCLES		1-BIN FIXED 2-POINTER 3-BIT(1)	ACRDRA ACRDR CENTFRL PRTASS1 PRTASSK PRTHDR PRTTRL RECLBUF	MONITOR	ROUTINE TO FIND CYCLES.
DATANME	FUNCTYPE	1-POINTER 2-CHAR(31)VAR		CHKSCON PALTSNM PRTHDRS STFLD WWARNP WWARN1	RETURNS DATAAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
DATATYP	FUNCTYPE	1-POINTER 2-CHAR(1)		ASUBST IFENAMS PRTMED RESIC SETTY SUBPROP SUPFORE UPDDIM WWARN1	RETURNS TYPE OF DATA NAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
DBLOFT	LEX				INITIALIZES SOME SWITCHES.
DIOPH		1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-BIN FIXED 5-BIN FIXED 6-BIT(1)		CHKCONT LOOPMV MAPV	OBTAINS THE SOLUTION OF THE DIOPHANTINE EQUATION A*X+B*Y=N, WHERE A,B, AND N ARE THE FIRST THREE ARGUMENTS. THE SOLUTION(XO,YO) IS RETURNED IN THE LAST TWO ARGUMENTS.
DUMPHEX		1-PTR 2-BIN FIXED	ASSMDH CONVPTR	ADJMAS MONITOR PRTADJL PRTASSX SVANAM TRACE	USED TO DUMP SPART OF MEMORY.
DUMPSET	DUMPHEX			MONITOR	USED TO GET THE ADDRESS OF THE MEMORY TO BE DISPLAYED
DUMPSYR	SVANAM				USED TO CLEANUP THE GARBAGE CAUSED BY A SYNTAX ERROR.
DUMPVL			ACRDR		DUMPS THE VAR-LIST STRUCTURE OF ALL DICTIONARY ENTRIES
DUPLDSK		1-POINTER 2-BIN FIXED	ACRDR ACRDRA ADJMRB ADJMSET DUPLSE ERSYST SETDTRP	RESIC	USED TO DUPLICATE A MEDIA STATEMENT

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
DUPLSE		1-POINTER 2-POINTER 3-POINTER	ACRDC IUSERD STORE ACRDUP2	DUPLDSK UPDDTRD	DUPLICATE A STORAGE ENTRY.
DUPLDUP			ACRDC ACRPOSR GINDEX2 GSBITS	ACRDICT	DUPLICATES DICTIONARY ENTRY FOR AN UPDATE DATA NAME.
ENDIDFT	LEX				PROCESSES THE "END" STATEMENT.
EQSUB		1-BIN FIXED 2-BIN FIXED *-BIT(1)		ANLIST	RETURNS('1'B) IF THE SUBSCRIPT NAMES REPRESENTED BY THE TWO ARGUMENT DICTIONARY INDICES ARE SAME.
ERNGIN1	JUNK			SSTORE	
ERNGTE	JUNK				
ERRSEM	SEMHELP			IFIELD IMEDIA ISPUNIT JUNK SSTORE	WRITES SEMANTIC ERROR MESSAGES.
ERSYST	ERROUT	1-CHAR(*)		ACRDUP ADJMNS CHKCONT CSUBLST DUPLDSK GCNODEL GDTAE GDTVI GVARDI IFENAMS LEXDUFN LIBADDM RETREVE SINITNM SSULRNG STFLD UPDASSN UPDDIM	REPORTS SYSTEM ERROR MESSAGES WITH A "SNAP".
FLAGSET	TRACE	1-CHAR(*)VAR		MONITOR	USED TO SET FLAG BITS (RUNTIME PARAMETERS).
FORCEPF	JUNK			GET_AE	
FREEPF				GAFSUBS	PROCEDURE TO FREE ARRAYS CREATED BY OBTAP.
FREEL	ANLIST	1-PTR		SUBPROP	USED TO FREE A LIST CREATED DURING SUBSCRIPT PROPAGATION.
GADFDIM		1-PTF *-BIN FIXED	GDINDEX	CSUBLST	RETURNS THE DIMENSION OF THE DATA NAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
GETVARS		1-BIN FIXED *-BIN FIXED	ACRPOSR	RICONSD	RETURNS THE NUMBER OF TARGET VARIABLE FOR THE ARGUMENT DICTIONARY ENTRY.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
GAPSEAS			FREERF GENDT GENSEA GNASSNM OSTAP UPDICT	RESICA	GENERATES ASSERTIONS FOR ELEMENTS OF DATA NAMES THAT ARE NOT DEFINED.
GANMFIV		1-BIN FIXED 2-CHAR(100) VAR	ACRDP ACRDPA GFONAME	RICONSE	RETURNS THE NAME OF THE ASSERTION THE TARGET VARIABLE OF WHICH IS GIVEN BY THE ARGUMENT DICTIONARY ENTRY.
GANMEV	GANMFIV	1-BIN FIXED 2-CHAR(100)VAR		WVARNT	RETURNS THE ASSERTION NAME WHICH USES THE ARGUMENT DICTIONARY ENTRY AS EITHER SOURCE OR TARGET VARIABLE.
GAGRPH	CLINC			CTHRES	
GENCYCL		1-BIN FIXED 2-BIN FIXED 3-PTR	ACRDPA ADJR ERSYST	GCYCLES	RETURNS ALL THE CYCLES THROUGH THE NODE GIVEN BY THE FIRST ARGUMENT.
GCYCLES		1-PTR	GCNODEL	CHKSCON	OBTAINS ALL CYCLES IN THE GRAPH (FOR SUBSCRIPT ANALYSIS)
GDIMEN		1-PTR 2-BIN FIXED	GDIMEN UPPTR	GDIMEN	RETURNS THE DIMENSION OF THE DATA NAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
GDINSEP		1-BIN FIXED 2-PTR	ACRDP	CSUPLST GROFDIM GENASSI GFAACTD GVARDI PRTDICT SETPR SETT9 SSUBRNG SUBPROP	RETURNS THE STORAGE ENTRY PTR ((ARGUMENT 2) OF THE DICTIONARY ENTRY IN ARGUMENT 1).
GDTAE		1-PTR 2-PTR 3-CHAR(1)	ACRDP ERSYST GDTVAR GDTVI	GENDT	CREATES A DERIVATION TREE FOR AN ARITH EXPRESSION.
GDTFC		1-PTR 2-PTR	GDTVI	GENDT	UPDATES THE "IF-OR-FOR-BOOL" DERIVATION TREE OF AN ASSERTION.
GDTAEIR	GDINDEX	1-PTR 2-PTR		SETT9 SUBPROP	RETURNS THE POINTER TO THE DAUGHTER LIST OF THE STORAGE ENTRY.
GDTVAR		1-PTR 2-BIN FIXED 3-PTR	GFONAME	GDTAE	OBTAINS A DERIVATION TREE FOR A VARIABLE

GDTVI		1-BIT(1)
		2-PTR
		3-PTR
		4-CHAR(1)
			ERSYST	GDTAE GDTFC	OBTAINS A DERIVATION TREE FOR AN INTEGER OR A VARIABLE.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
GENASSI		1-PTR 2-CHAR(1)	ACRDD ACRSTVS ADJMSET GDINDEX GETINAM GFLDTYP IUSERA RETRPL SCONSET STORE WWARN3 WWARN4	ANALIND	GENERATES ASSERTION FOR A PART OF THE DERIVATION TREE IN AN ASSERTION.
GENDFAP	JUNK				
GENDT		1-PTR 2-PTR	GDTAE GDTFC	GAFSUBS	GENERATES A DERIVATION TREE FOR AN ASSERTION.
GENI	GENSTMT	1-CHAR(*) 2-CHAR(*)VAR 3-CHAR(*)VAR 4-CHAR(*)VAR 5-(*)BIN FIXED		ACRPTR	GENERATES DD STATEMENT FOR THE INTERIM DATA NAME (ARGUMENT 1).
GENMF	GENSTMT	1-CHAR(*)		RESIC	GENERATES MEDIA OR FILE STATEMENT FOR THE ARGUMENT NAME.
GENKST	ACRPTR	1-CHAR(*)VAR 2-CHAR(*) 3-BIN FIXED 4-PTR			USED GENERATE STORAGE ENTRIES FOR THE KEYWORD NAMES.
GENSFA		1-PTR 2-PTR 3-PTR		GAFSUBS	CREATES THE DATA AREA FOR AN ASSERTION.
GENSTMT		1-CHAR(*) 2-CHAR(*) VAR 3-CHAR(*)VAR 4-CHAR(*)VAR	ACRDD IMEDIA LISTCET LISTSL PRTASSI PRTSTMT SCONSET STORE WSAMOUT WWARN4	ACRCMST ACRDICT ACRPTR RESIC	GENERATES A DD STATEMENT
GENSUB	GENSTMT	1-CHAR(*)		GSUBAPT SETSUBN SUBSCRT	GENERATES A SUBSCRIPT STATEMENT FOR THE ARGUMENT NAME.
GET_AE		1-PTR 2-CHAR(1)	ANALIND CONVPTR FOPCOMB SUBSCRT	ACRSTVS	OBTAINS ALL THE NAMES IN THE DERIVATION TREE OF AN ARITHMETIC EXPRESSION
GETAEN	GENASSI	1-PTR			RETURNS THE VARIABLE NAME REPRESENTED BY THE ARGUMENT ARITH EXPRESSION.
GETQSNM	GETQNM	1-POINTER 2-CHAR(100)VAR			RETURNS THE FULLY QUALIFIED NAME OF A STORAGE ENTRY (AFTER REMOVING ALL BLANKS AND PREFIXES).

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
GETPRSN		1-PTR 2-(*)PTR 3-(*)BIN FIXED 4-BIN FIXED			RETURNS IN THE SECOND ARGUMENT, ALL THE STORAGE ENTRY POINTERS OF ALL REPEATING PARENT DATA NAMES OF THE STORAGE ENTRY GIVEN BY THE FIRST ARGUMENT.
GETFNAM		1-CHAR(*) 2-BIN FIXED	ACRPP RETRNAM	ACRADJM	OBTAINS THE DICTIONARY ENTRY FOR THE ARGUMENT NAME (WHICH IS USED WITH SOME PREFIX LIKE POINTER, EXIST, AND SO ON).
GETINAM	GETFNAM	1-PTR		GENASSI	OBTAINS A NEW INTERIM NAME.
GETKEYT	KEYPTR	1-BIN FIXED 2-PTR			RETURNS IN 2ND ARGUMENT, THE STORAGE ENTRY POINTER OF THE KEYWORD, THE INDEX OF WHICH IS GIVEN BY THE FIRST ARGUMENT.
GETPTR	PRTAB	1-CHAR(*) 2-PTR			RETURNS A POINTER FOR THE ARGUMENT CHARACTER STRING.
GETQNM		1-PTR 2-CHAR(*) 3-CHAR(100)VAR		ACRDUP ASUBST	RETURNS A QUALIFIED NAME FROM A STORAGE ENTRY(ARGUMENT 1).
GETQNPT	GETQNM	1-PTR 2-CHAR(*) 3-BIN FIXED 4-CHAR(100)VAR		ACRPTR	RETURNS A QUALIFIED NAME FROM A STORAGE ENTRY REPRESENTED BY THE FIRST ARGUMENT.
GETRPRN		1-PTR 2-PTR		ASUBST GEXACTD	OBTAINS THE PARENT (OR ITSELF) OF THE FIRST ARGUMENT POINTER THAT HAS REPEATING STRUCTURE.
GETSIZE		1-PTR 2-PTR		GRNGDIM	RETURNS A POINTER TO A STRUCTURE THAT DEFINES THE DIMENSION OF THE NAME REPRESENTED BY THE ARGUMENT STORAGE ENTRY POINTER.
GETVPTR		1-CHAR(*) 2-PTR			RETURNS POINTER TO THE "VARIABLE" ENTRY OF THE DIRIVATION TREE FOR THE ARGUMENT NAME.
GEXACTD		1-PTR 2-BIN FIXED 3-PTR 4-BIN FIXED	GDINDEX GETRDNM GPPTR	CSUBLST IFENAMS	USED TO OBTAIN THE DIMENSION OF THE DATA NAME WITH WHICH A SUBSCRIPT NAME IS ASSOCIATED.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
GFLDTYP	GENSTAT	1-PTR *--CHAR(1)		GENASSI	RETURNS THE FIELD TYPE OF THE ARGUMENT STORAGE ENTRY.
GFQNAME		1-PTR *--CHAR(100)VAR	DECLSUF	GANMFTV PRTQNM RICONSE SUBPROP	RETURNS THE FULLY QUALIFIED NAME FOR A STORAGE ENTRY PTR.
GFQNAM1	CFQNAME	1-PTR *--CHAR(100)VAR		GDTVAR	RETURNS THE FULLY QUALIFIED "KEY-NAMES" FOR THE ARGUMENT STORAGE ENTRY.
[illegible]	GENASGT	1-PTR 2-PTR		RESIC	GENERATES AN ASSERTION EQUATING THE DATA NAMES REPRESENTED BY THE TWO ARGUMENTS.
[illegible]	GENASGT	1-PTR		RESIC	GENERATES AN ASSERTION BY EQUATING THE NAME REPRESENTED BY THE ARGUMENT WITH "" OR 0.
GANDEX2	CDINDEX	1-PTR *--BIN FIXED		ACRDO DUPLUDE	RETURNS "INDEX2" ENTRY CORRESPONDING TO THE ARGUMENT STORAGE ENTRY POINTER.
GNAPV		1-BIN FIXED 2-BIN FIXED 3-PTR	ACRPOSR	CSMAPV CTHRES RESIC8	RETURNS THE POINTER TO THE MAP-VECTOR (IN THIRD ARGUMENT) ASSOCIATED WITH THE TYPE-8 EDGE BETWEEN THE TWO NODES (REPRESENTED BY THE FIRST TWO ARGUMENTS)
GNASSN	GENASST			GAFSUBS	OBTAINS A NEW ASSERTION NAME.
GPICPTR	DECPIC	*--PTR		STFLD	RETURNS THE POINTER TO THE PICTURE SPECIFICATION.
GPINDEX	CDINDEX	1-BIN FIXED 2-BIN FIXED		RICOMPE UPDICT	RETURNS THE PRIMARY DICTIANARY INDEX OF THE ENTRY REPRESENTED BY THE FIRST ARGUMENT.
GPINSAT	CDINDEX	1-PTR 2-BIN FIXED			RETURNS THE PRIMARY INDEX OF THE STORAGE ENTRY (1ST ARGUMENT).
GPLINE#	NODTYPE	1-BIN FIXED *--BIN FIXED		SUBPROP	RETURNS THE STATEMENT NUMBER OF THE DICTIONARY ENTRY REPRESENTED BY THE FIRST ARGUMENT.
GPPTR	CDINDEX	1-PTR *--PTR		CSUBLST CSUBRED GDIMEN GFXACTD RESIC SSUBRN.	RETURNS THE PARENT STORAGE ENTRY PTR OF THE ARGUMENT STORAGE ENTRY.
GREDFUN		1-BIN FIXED *--BIT(1)	ACRDR		RETURNS "1"B IF THE ARGUMENT DICTIONARY ENTRY IS USED AS AN ARGUMENT OF A REDUCTION FUNCTION.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
GRNGDIM			ACRPOSR GETSIZE UPVLIST	RFSIC	OBTAINS RANGE AND DIMENSION OF EACH DICTIONARY ENTRY.
GSBITS	GEINDEX	1-PTR 2-BIT(*)		ACRDC DUPLUDE	RETURNS ALL THE STATUS BITS OF THE STORAGE ENTRY(ARGUMENT 1).
GSFACT	CLINC			CTHRES	
GSTBITS	GSINDEX	1-PTR 2-BIN FIXED 3-BIT(1)		PRTHDRS	RETURNS I TH (ARGUMNET 2) OF THE STORAGE ENTRY(ARGUMENT 1).
GSTEPTR		1-CHAR(*) VAR 2-PTR	ACRMP RETRNAM	ASUBST IFENAMS	RETURNS THE STORAGE ENTRY PTR FOR THE ARGUMENT NAME.
GSUBSPT		1-PTR	GENSUB	IFENAMS RICOMPE	GENERATES A DEFAULT SUBSCRIPT STATEMENT.
GTVARPTS	GTVARS	1-(*)PTR		RICONSS	RETURNS ALL THE TARGET VARIABLE POINTERS CORRESPONDING TO A VARIABLE NAME.
GVARDIC		1-PTR 2-CHAR(1) 3-BIN FIXED	ENSYST GDINDEX RETREVE RETRAL	SUBPROP	RETURNS THE DICTIONARY OF THE STORAGE ENTRY, THE NAME OF WHICH IS GIVEN BY THE FIRST ARGUMENT POINTER.
GVARPTR		1-PTR 2-PTR		IFENAMS	CREATES A "VARIABLE" STRUCTURE FOR THE DATA NAME REPRESENTED BY THE STORAGE ENTRY (FIRST ARGUMENT) AND RETURNS POINTER TO THAT STRUCTURE.
IFENAMS			ACRDC ACRDR ACRDFA ACRPOSR ADJMSET DATAYPE ENSYST GEXACTD GSTEPTR GSUBSPT GVARPTR SETDTRP SSUBRNG UPDASSN	RESIC	IDENTIFIES EACH FOR-EACH NAME WITH A SUBSCRIPT NAME.
IFFATLN	IFIELD				SAVES THE LENGTH OF FIXED DECIMAL FIELD (OR INTERIM).
IFFATPR	IFIELD				SAVES PRECISION OF A FIXED DECIMAL FIELD
IFIELD			ENRSEP SINCRUN		INITIALIZES THE DATA AREA FOR FIELD, GROUP, ... STATEMENTS (SAP -SUPPORTING ROUTINE).
IFOCCL	IFIELD				SAVES THE LOWER BOUND OF A REPEATING DATA NAME.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
IFUCCII	IFIELD				SAVES THE UPPER BOUND OF A REPEATING DATA NAME.
IFJCCN	IFIELD				SAVES THE LOWER BOUND OF A REPEATING DATA NAME.
IF[illegible]CCN	IFIELD				SAVES THE UPPER BOUND OF A REPEATING DATA NAME.
I[illegible]CCS	IFIELD				SAVES THE LOWER BOUND OF A REPEATING DATA NAME.
IFCCCJS	IFIELD				SAVES THE UPPER BOUND OF A REPEATING DATA NAME.
IFPAYE	IFIELD				PREFIXES THE NEXT DATA NAME IN THE INPUT BY "#E".
IFPAYL	IFIELD				PREFIXES THE NEXT DATA NAME IN THE INPUT BY "#L".
I[illegible]	IFIELD				USED TO INITIALIZE "PREFIX" TO "#L".
I[illegible]	IFIELD				INDICATES THAT LENGTH OF A FIELD WAS GIVEN BY "*".
IFSVAR	IFIELD				SETS "VARYING-BIT" FOR CHARACTER STRING FIELDS.
I[illegible]LK	IMEDIA			IMEDIA1	OBTAINS BLOCKSIZE PARAMETER.
IMDEN	IMEDIA	M-BIT(1)		IMEDIA1	OBTAINS DENSITY PARAM.
IMD[illegible]	IMEDIA				USED TO INITIALIZE MEMBER OF DS NAME.
IMD[illegible]1	IMEDIA				USED TO UPDATE MEMBER OF DS NAME.
IMDSN	IMEDIA				OBTAINS DATA SET NAME.
IMDSNF	IMEDIA				USED TO INITIALIZE DS NAME.
IMDSNN	IMEDIA				USED TO UPDATE DS NAME.
IMEDIA			CONVF CONVAX CONVFY ERRSEM LEX LEXENAB RSKIPTB	ACRDICT GENSTMT	INITIALIZES DATA ARE FOR MEDIA AND FILE STATEMENTS.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
IMLINE	IMFDIA			RMEDIAT	OBTAINS LINESIZE PARAMETER.
IMNOTR	IMFDIA	*--BIT(1)		RMEDIAT	OBTAINS NUMBER OF TRACKS PARM.
IMORG	IMFDIA	*-- IT(1)		RMEDIAT	OBTAINS ORGANIZATION TYPE.
IMPAGE	IMFDIA			RMEDIAT	OBTAINS PAGESIZE PARAMETER.
IMREC	IMFDIA			RMEDIAT	OBTAINS RECORDSIZE PARAMETER.
IMSFILE	IMFDIA			RMEDIAT	OBTAINS START FILE # (FOR A TAPE).
IMANTR	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS # OF TRKS PARM.
IMABLK	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS BLOCKSIZE
IMADEN	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS DENSITY PARAM.
IMADSN	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS DS NAME.
IMALIN	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS LINESIZE FOR PRINTING.
IMAORG	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS ORGANIZATION PARM.
IMAPAG	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS PAGESIZE PARAM.
IMAREC	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS RECORDSIZE PARM.
IMASF	IMFDIA	*--CHAR(100)VAR		PRTSTMT	RETURNS START FILE PARM.
ISPDIM	ISPUNIT				SAVES COMPONENTS OF SPACE PARM.
ISPINC	ISPUNIT				SAVES COMPONENTS OF SPACE PARM.
ISPINIT	ISPUNIT				SAVES COMPONENTS OF SPACE PARM.
ISPRLSE	ISPUNIT				SAVES COMPONENTS OF SPACE PARM.
ISPTAB	ISPUNIT				SAVES TAB PARAMS.
ISPTALE	ISPUNIT				SAVES TAB PARAMS.
ISPTAHI	ISPUNIT				SAVES TAB PARAMS.
ISPUNIT			CONV# ERRSEM		CONTAINS SOME ADDITIONAL SUPPORTING ROUTINE FOR SAP FOR SAVING ARGUMENTS OF FILE AND MEDIA STATEMENTS.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
ISPASPC	ISPUNIT			PRTSTMT	SAVES COMPONENTS OF SPACE PARM.
ISPATAB	ISPUNIT			PRTSTMT	SAVES TAB PARAMS.
IOCHARC	IOUNIT			RMEDIAT	SAVES CHARACTER CODE.
IODISP	IOUNIT			RMEDIAT	SAVES DISPOSITION.
IOELABL	IOUNIT			RMEDIAT	SAVES EXTERNAL LABEL.
IOILABL	IOUNIT			RMEDIAT	SAVES INTERNAL LABEL.
IOLABEL	IOUNIT			RMEDIAT	SAVES LABEL TYPE.
IOUNIT		--BIT(1)	CONVRY CONVRZ LEX LEXBUFN LEXENAB	RMEDIAT	CONTAINS SOME ADDITIONAL SUPPORTING ROUTINE FOR SAP FOR SAVING ARGUMENTS OF FILE AND MEDIA STATEMENTS.
IOPARTY	IOUNIT	--BIT(1)		RMEDIAT	SAVES PARITY.
IORECFM	IOUNIT	--BIT(1)		RMEDIAT	SAVES RECORD FORMAT.
[illegible]	GENSTMT			GENASSI UPDICT	OUTPUTS A MESSAGE TO SAPOUT.
[illegible]	GENSTMT			ACRDICT DUPLSE	OUTPUTS A MESSAGE TO SAPOUT.
[illegible]	IOUNIT			STFLD	USED TO INITIALIZE INT & EXT LABELS.
IOXCC	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS CHAR CODE.
IOXDSP	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS DISPOSITION OF FILE.
IOXEL	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS EXTERNAL LABEL.
IOXIL	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS INTERNAL LABEL.
IOXPAR	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS PARITY
IOXRFM	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS RECORD FORMAT.
IOXTLB	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS TAPE LABEL.
IOXUNT	IOUNIT	--CHAR(100) VAR		PRTSTMT	RETURNS UNIT-TYPE.
JUNK			ERRSER		

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
KEYWRD		1-PTR 2-BIT(1)	RECLBUF	RESIC	RETURNS "1"B IF THE STORAGE ENTRY REPRESENTED BY THE ARGUMENT POINTER REPRESENTS A KEYWORD.
LCM		1-BIN FIXED 2-BIN FIXED *-BIN FIXED	GCD	RANGLIT	RETURNS "LEAST COMMON MULTIPLE".
LEX			LEXPOPN LEXREAD PRTAM SCOMAUD	IMEDIA IUNIT LEXEOFN LEXREAD RECFLOT RECNAME RECPIC RMEDIAT RPAGE SVANAM TINPRNT	LEXICAL ANALYZER.
LEXBIT	LEXREAD	*-BIT(1)			RECOGNIZES A BIT STRING.
LEXGLEN		1-BIN FIXED *-CHAR(31)VAR	ERSYST LEX LEXENAB RSKIPTO	IUNIT RECFLOT RPAGE SVANAM	RETURNS THE N TH TOKEN IN THE INPUT.
LEXCSET	LEXREAD	1-CHAR(*)		RECPIC SVANAM	RETURNS THE CHARACTER STRING RECOGNIZED.
LEXCLEN	LEXREAD	*-BIN FIXED		RECPIC SVANAM	RETURNS THE LENGTH OF THE CHARACTER STRING RECOGNIZED.
LEXCST	LEXREAD	*-BIT(1)		RECPIC	RECOGNIZES A CHARACTER STRING.
LEXDSAB	LEX			TINPRNT	DISABLES THE LEX.
LEXENAB	LEX			IMEDIA IUNIT LEXEOFN LEXREAD RCONST RECFLOT RECNAME RMEDIAT TINPRNT	ENABLES LEX
LEXFAIL	LEX			SPOPF	USED TO PROCESS SYNTAX ERRORS.
LEXGT	LEX	*-BIN FIXED		SPOPF LEXREAD	RETURNS THE COLUMN NUMBER OF THE NEXT INPUT TOKEN.
LEXGTS	LEX	*-CHAR(31)VAR		SPOPF	RETURNS THE NEXT CHARACTER IN THE INPUT.
LEXCPOS	LEXREAD	*-BIN FIXED(31)			RETURNS THE COLUMN NUMBER OF THE NEXT CHARACTER IN THE INPUT.
LEXPOPN	LEXBUF*			LEX	POPS A TOKEN IN THE STACK LEXBUFS.
LEXREAD			LEX LEXENAB LEXGT LEXRI WSAPOUT	LEX	READS MODEL STATEMENTS.
LEXRI	LEX	1-BIN FIXED		LEXREAD	RESETS THE COLUMN IN THE INPUT (TO BE SCANNED).

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
LEXSINC	LEX				INCREMENTS THE CURRENT STATEMENT NUMBER.
LEXSREF	LEXPREAD	1-PTR		SVREF	USED TO REFER A SECTION FROM THE MODEL STMT. DATA BASE.
LEXTEMPQ	LEXBUFS				SAVES A QUALIFIED NAME IN THE LEXBUFS STACK.

LIBADDM		1-CHAR(1)
		2-CHAR(3)
		3-PTR
		4-FILE
			ESSYST LIBREFS WROUT	LIBU	ADDS A MEMBER TO MODEL STATEMENT DATA BASE

LIBCR		1-CHAR(150) VAR			CREATES THE MODEL STATEMENT DATA BASE.
LIBLMEM	LIBADLY	1-CHAR(*) 2-FILE			LISTS MEMBER NAMES.
LIBPRALL	LIBADLY				PRINTS ALL MEMBERS IN THE LIBRARY.
LIBREAD	SVREF				READS A SECTION FROM MODEL STATEMENT DATA BASE.
LIBREFS		1-CHAR(*)VAR 2-PTR	RSKIPTR	LIBADDM LIBU SVREF	REFERS A SECTION OF MODEL STATEMENT DATA BASE.
LIBU			LIBADDM LIBREFS RSKIPTE RSKIPLR WROUT		PROCEDURE TO UPDATE MODEL STATEMENT DATABASE.
LINKSEG			RETRIEVE RETRVE	ACRDICT	LINKS SIMULTANEOUS ASSERTIONS IN THE ASSOCIATIVE MEMORY
LISTGET	LISTS:	1-CHAR(*)		CHKSCON GENSTMT PRTASSX PRTHDRS PRTSTMT RICONSL SUBPROP WWARN1 XWRITE1 XWXREF	RETURNS IN THE ARGUMENT, THE STRING IN THE PRINT STACK.
LISTPA	LISTS:	1-CHAR(*)VAR		CHKSCON PALTSNM PRTASSX PRTHDRS PRTQNM PRTSTMT RICONSE SUBPROP WWARN1 XWGATTR XWSIZE XWXREF	ADDS A STRING TO THE PRINT STACK.
LISTSI				PRTASSX PRTHDRS PRTSTMT RICONSE SUBPROP XWRITE1 XWXREF	INITIALIZES PRINT STACK.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
LISTSL	LISTSI	*-BIN FIXED		CHKSCON GENSTMT PRTASSX PPTHDRS PRTSTMT RICONSE SUBPROP WWARNT XWRITE1 XWXREF	RETURNS THE LENGTH OF PRINT STACK.
LOGPMV		1-(*,*)BIN FIXED 2-(6)BIN FIXED	DIGPH WEIGHTV	CSMAPV	COMPUTES THE SELF MAP VECTOR.
MAPV		1-(3)BIN FIXED 2-(3)BIN FIXED 3-(*) BIN FIXED	DIGPH WEIGHTV	RANGEM RANGLIT RICONSS	COMPUTES THE MAP VECTOR BETWEEN THE TWO NODES (REPRESENTED BY THE FIRST TWO ARGUMENTS).
MERGEC		1-PTF	ACRDR RSUBPTG SSUBFTR	ACRSTVS SUPFORE	
MONITOR		1-CHAR(100)VAR	ACRADJM ACRDICT BUILDD CYCLES DUMPHEX DUMPSET FLAGSET PRTADJM PRTAM REMOVEM RESIC RESICS SAP SCOMINT SUBPROP WSAPOUT XREFPRT CHKSCON PRTADJL PRTLBUF		THE MAIN ROUTINE THAT EXECUTES THE DIFFERENT PHASES OF THE MODEL PROCESSOR.
NODTYPE		1-BIN FIXED 2-BIT(1) 3-CHAR(1)	ACRDR	REMOVES SUBPROP	RETURNS NODETYPE.
NODTYPN	NODTYPE	1-BIN FIXED 2-BIN FIXED 3-BIT(1)		SUBPROP	RETURNS I TH BIT (ARGUMENT 2) FROM STATUS BITS CORRESPONDING TO THE FIRST ARGUMENT DICTIONARY ENTRY.
NULLPOS		1-PTR		ACRDU	
OBTAP			ACRDRA ADJARS SENTRYR TESTMED	GAFSUBS RESICE	OBTAINS A SET OF VECTORS THAT CONTAIN # OF PREDECESSORS, #OF SUCCESSORS ETC.
PALTSCR		1-PTF	CONVP DATAARE LISTSA ROOTPTR	PRTGNP PRTSTMT WWARNT	PRINTS ALTERNATE NAME(USING "FOREACH") FOR A SUBSCRIPT NAME.
PRTAP	LSTAP			REMOVES	LISTS THE VECTORS CREATED BY THE ROUTINE OBT#P.
PRTADJL			ACRPOSR DUMPHEX	MONITOR	PRINTS THE PRECEDENCE MATRIX (LIST FORM)
PRTADJM	PRTDICT			MONITOR	PRINTS ADJACENCY MATRIX.
PRTAEA	PRTASUX	1-PTP			PRINTS AN ARITH. EXPRESSION

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
PRTAM		1-PTR	CONVPTR RECLBUF	LEX MONITOR RETREVE	DUMPS THE ASSOCIATIVE MEMORY
PRTASSN	PRTDDS			PRTASSX	PRINTS A FRAME FOR ASSERTIONS.
PRTASSX			CONVB DUMPHEX LISTGET LISTSA LISTSI LISTSL PRTASSN PRTNM PUTLINE RECLBUF RETREVE RETAPE SORTASS WROUT	CYCLES	GENERATES A LISTING OF ALL ASSERTIONS.
PRTASST	PRTASSV	1-PTR		CYCLES WWARN1	PRINTS AN ASSERTION
PRTATTS	PRTSTPT	1-PTR		XVGATTR	RETURNS THE ATTRIBUTES OF A STATEMENT.
PRTCCM		1-PTR 2-BIN FIXED	PUTLINE	PRTMED	PRINTS COMMENT LINE.
PRTDA	PRTDICT				DUMPS DATA AREA OF THE MODEL SPECIFICATION.
PRTDDS			CENTERL PRTHDR PRTTPL PRTHDRO PRTTPLO CENTERLO	PRTMED	PRINTS FRAME FOR DD STATEMENTS
PRTDICT			ACRDFP ACRPOS GDINDEX PRTHEXC RECLBUF		PRINTS THE HEXADECIMAL DUMP OF THE DICTIONARY
PRTFDS	PRTDDS			PRTMED	PRINTS A FRAME FOR FILE STATEMENTS.
PRTHDR			PUTLINE	CYCLES PRTDDS PRTHDRS	USED TO PRINT FRAMES IN PRTMED.
PRTHDRO	PRTHDR	1-BIN FIXED		PRTDDS	USED TO PRINT A FRAME FOR FILE STATEMENTS.
PRTHDRS			CENTERL DATANME GSTBITS LISTGET LISTSA LISTSI LISTSL PRTHDR PRTTPL PUTLINE RETREVE RETRWE RSKIPTB	PRTMED	PRINTS HEADER SECTION OF THE MODEL SPEC.
PRTHEX	DUMPHEX	1-PTR 2-BIN FIXED			DUMPS MAIN MEMORY
PRTHEXC	DUMPHEX	1-PTR 2-BIN FIXED		ADJMAP PRTDICT TRACE	DUMPS MAIN MEMORY
PRTIDS	PRTDDS			PRTMED	PRINTS A FRAME FOR INTERIM STATEMENTS.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
PRTLFLF	RECNAME			MONITOR	DUMPS ALL LONG NAMES (LENGTH > 8) USED IN THE MODEL SPECIFICATION.
PRTPSL		1-BIN FIXED	DATAYPE PRTCOM PRTDDS PRTHDRS PRTIDDS PRTSTMT PUTLINE RETREVE RETRFE PRTFDS PRTSDDS		GENERATES THE FORMATTED LISTING OF DD STATEMENTS.
PRTPTR	PRTAP	1-CHAR(*) 2-PTR		RETREVE	PRINTS A HEXDECIMAL REPRESENTATION OF A POINTER.
PRTQNM		1-PTR 2-PTR	ACPDR GFGNAME LISTSA RECLBUF PALTSNM RETREVE RETRFE	PRTASSX	PRINTS THE QUALIFIED DATA NAMES
PRTQNMT	PRTQNP	1-PTR		SUOPNOP	PRINTS A QUALIFIED NAME THE DERIVATION TREE FOR WHICH IS GIVEN BY THE ARGUMENT POINTER.
PRTSEDS	PRTDDS			PRTMED	PRINTS A FRAME FOR SUBSCRIPT STATEMENTS.
PRTSTMT		1-PTR 2-BIN FIXED	CONVP IXXBLK IXXDEN IHXDSN IHXPTR IXXLIN INXORG INXPAG INXREC INXSF ISFXSPC ISPATAP IUXCC IUXDSP IUXEL IUXIL IUXPAR IUXRFM IUXTLP IUXUNT LISTGET LISTSA LISTSI LISTSL PUTLINE RECLBUF RETRFE PALTSNM	GENSTPT PRTMED WWARNT	PRINTS A PARTICULAR DD STATEMENT (GIBEN BY THE ARGUMENT STORAGE ENTRY PTR).
PRTTFL	PRTHDR			CYCLES PRTDDS PRTHDRS	USED IN PRINTING A HEADER.
PRTTFLD	PRTHDR	1-BIN FIXED		PRTDDS	USED TO PRINT A FRAME FOR FILE STATEMENTS.
PUTLINE		1-CHAR(*)VAR 2-BIN FIXED 3-BIN FIXED		PRTASSX PRTCOM PRTHDR PRTHDRS PRTMED PRTSTMT	OUTPUTS THE ARGUMENT STRING TO FILE "OUTM", WITH APPROPRIATE INDENTATION, IF THE LENGTH OF STRING IS > 72.
RANGPLC		1-BIN FIXED 2-BIN FIXED 3-BIN FIXED 4-PTR 5-BIN FIXED		RANGLIT	USED TO CREATE A NEW RANGE LIST DURING SPLITTING OF SUBSCRIPT RANGES.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
RANGEP		1-PTR 2-PTR *-BIT(1)	MAPV	RANGSR	USED IN SUBSCRIPT RANGE SPLITTING.
RANSLIT			LCM MAPV RANGALC	RANGSR	SPLITS A SUBSCRIPT RANGE.
RANGSET		1-BIN FIXED	ACPDR	RFSICR	CREATES A SUBSCRIPT RANGE LIST.
RANGSB		1-PTR	RANGLIT RANGEM	RFSICR	USED TO SUBTRACT A RANGE FROM ANOTHER.
RDLARAY	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RECCRE	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RECONST		*-BIT(1)	RECLBUF LEXENAB		RETURNS "1"B IF THE TOKEN REPRESENTS A CONSTANT (LIKE, "TRUE", "FALSE").
[illegible]	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RDIST	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
[illegible]	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RECCPT	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
[illegible]	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RECFINT	RECNAME	*-BIT(1)			RECOGNIZES INTEGER CONSTANTS.
RECFLOT		*-BIT(1)	LEX LEXEUFN LEXENAB		RECOGNIZES FLOATING POINT DATA (SYNTAX ANALYSIS)
RECHELP	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RECINT	RECNAME	*-BIT(1)		RMEDIAT RPAGE	RECOGNIZES INTEGER CONSTANTS.
RECIT	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RECKEYN	RMEDIAT	*-BIT(1)			RECOGNIZES KEY NAMES (SYNTAX ANALYSIS).
RECLELF	RECNAME	1-CHAR(*)VAR *-CHAR(31)VAR		$POPF BUILDD CYCLES GFGNAME KEYWORD PRTAM PRTASSX PRTDICT PRTONM PPTSTMT RCONST STFLD SUBSCRT TESTFUN XWGATTR XWRITE1 XWSIZE	RETURNS THE FULL NAME (UNCODED) NAME FOR A KEYNAME.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
RECNAME		*-BIT(1)	LEX LEXENAB RSKIPTR	RMEDIAT TINPHNT	RECOGNIZES DATA NAMES (SYNTAX ANALYSIS).
RECGF	PPAGF	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RECPIC		*-BIT(1)	LEX LEXCGET LEXCLEN LEXCSTR		RECOGNIZES PICTURE ATTRIBUTES.
RECSTW	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RECTC	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
REMDTRP		1-PTR 2-PTR		CSUBLST	REMOVES THE PARENT-DAUGHTER RELATIONSHIPS BETWEEN THE TWO ARGUMENT STORAGE ENTRIES.
REMOVEP			ACRDRA ADJRPS NODTYPE ENTMP SENTRYR TESTMED	MONITOR	REMOVES TYPE-8 EDGES THAT CAUSES CYCLES.
REMOVDL	ANLIST	1-PTR			REMOVE DUPLICATE SUBSCRIPTS IN A LIST.
RESIC			ACRDD ACRDFP ACRDPA ADJX ADJRSET DUPLDSK GENMF GENSTAT GIMPA GIMPI RETREVE RETRPE PETRNAM SETDTMP CSUBLST DATATYPE GPPTR GRNGDIM IFENAMS KEYWORD RESICS SETWOFE SETTY SUPFORC	MONITOR	RESOLVES INCOMPLETENESS
RESICA			GAFSUBS GRAPV OBTAP RANGSET RANGSR RICOMPF ACRDR	MONITOR RESIC	RESOLVES THE INCOMPLETENESS IN THE DEFINITION OF ELEMENTS OF AN ARRAY.
RETPTRS	CONVERT				
RETARE		1-CHAR(*) *-BIT(1)		ACRCHST ACRDICT ACRDUP ASUPST CSUBLST GENASSI GVARDI LINKSEQ PRTASSA PRTHDRS PRTMED PRTGNM PRTSTPT RESIC RSUPPT6 SETSUBN SETT56 UPDDTR	RETURNS THE NUMBER OF STORAGE ENTRIES FOR THE ARGUMENT DATA NAME.
RETNAM	RETARE	1-CHAR(*) 2- BIN FIXED			RETURNS NUMBER OF NAMES.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
RETRSVF		1-CHAR(*) VAR 2-CHAR(*)VAR 3-PTR 4-(*)PTR 5 BIN FIXED	ENSYST PRTAM PRTPTR	ACRCHST ACRDICT ACRDUP ASUBST CSUBLST GVARDI LINKSEQ PRTASSX PRTHDRS PRTMED PRTONM RESIC RSUBPTG SETSUBR SETTS6 SUBSCRT SVSTV UPDDTR WMESSAG	RETREVES STORAGE ENTRIES FROM THE ASSOCIATIVE MEMORY.
RETRSAM	RETRVE	1-CHAR(*) (*)CHAR(10) 3-BIN FIXED		ACRCHST ACRDICT ACRDUP GETFNAM GSTEPTR RESIC SUBSCRT	RETURNS ALL DIFFERENT REPRESENTATIONS FOR THE SAME NAME.
RETRFFX	RETRVE	1-CHAR(*) (*)CHAR(10) 3-BIN FIXED		ACRDICT	RETURNS ALL NAMES HAVING SAME PREFIX.
RETTLIT	INPC				
REXTNSP	SPACE	*-BIT(1)		RMEDIA1	USED TO RECOGNIZE SAP KEYWORDS.
PFIFLDT	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
PFILE	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
PFILES	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
PFLDFEC	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
PFLDTMP	PPAGE	*-BIT(1)		SVANAM	USED TO RECOGNIZE SAP KEYWORDS.
RICOMPE		1-BIN FIXED	ACPD1 ACPDR ADJMSET GSUBNFT SETTYPE GPINDEX GEXACTP	RESIC8	REPORTS INCOMPLETENESS DUE TO UNDEFINED ARRAY ELEMENTS.
RICONSE		1-PTR 2-PTR 3-(*,*)BIN FIXED	ACPDR CONVM GANMFTV GFONAME LISTGET LISTSA LISTS1 LISTS2 WWARN	RICONS8	REPORTS ERRORS DUE TO MULTIPLE DEFINITION OF DATA NAMES.
RICONS8			GETVARS GTVARPS MAPV RICONSE		CHECKS FOR MULTIPLE DEFINITION OF DATA ELEMENTS.
RINTNAM	PPAGE	*-BIT(1)		RMEDIA1	USED TO RECOGNIZE SAP KEYWORDS.
RISAFE	PPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RKEY	PPAGE	*-BIT(1)		RMEDIA1	USED TO RECOGNIZE SAP KEYWORDS.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
RKAREF	RPAGE	*-BIT(1)		TINPANT	USED TO RECOGNIZE SAP KEYWORDS.
RKASFILL	RPAGE	*-BIT(1)		TINPANT	USED TO RECOGNIZE SAP KEYWORDS.
RLINE	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RLOSECYC	CYCLES				RELEASE STORAGE ALLOCATED BY CYCLES ROUTINE.
RMEDIA	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RMEDIAT			RNOTFAT RORGN RPAGE RPARITY RRECFM RRECNAM RSFILE RTLABEL RUNIT SINCHNK XVBLK XVREC		
RMEDI-1		*-BIT(1)	IMBLK IMDEN IMLINE IMNOTR INORG IMPAGE IMREC IMSFILE IUCHARC IUDISP IUELABL IUILABL IULABEL IUNIT IUPARTY IURECFM LEX LEXENAB RBLKNAM RCCODE RDENSITY RDISP RDSNAME RECINT RECNAME REXTNAM RINTNAM RKEY RLINE		RECOGNIZES SOME OF THE KEYWORDS IN FILE AND MEDIA STATEMENTS.
RMEDIA2	RMEDIAT	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RMODULE	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RMOUNT	RMEDIAT	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RNOTFAT	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RNOTPTR	SNCHFIL	1-PTR		PALTSNM SUBPRC	OBTAIN THE ROOT STORAGE PTR OF A DATA NAME REPRESENTED BY THE ARGUMENT PTR.
RORGN	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RPAGE		*-BIT(1)	LEX LEXJUFN RECINT XVBLK XVCHARC XVDEN XVDISP XVDSN XVFLABL XVILABL XVKEY XVLABEL XVLINE XVNOTR XVORG XVPAGE XVPARTY XVREC XVRECFM XVSFILE XVSUNIT XVTAB XVUNIT	RMEDIAT	USED TO RECOGNIZE A SAP KEYWORD.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
RPARITY	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RPICSEC	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RPLCTR	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RRECTR	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RRECMAX	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
RRESERV	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RRELSEC	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RRLSE	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RSEC	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RSFILE	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
PSKIPL	CONV	1-CHAR(*) *-CHAR(100)VAR		LTHU	SKIP LEADING BLANKS
PSKIPTR	CONV	1-CHAR(*) *-CHAR(100)VAR		IMEDIA LEXRUFN LIBREFS LIBU PRTHDRS RECNAME	SKIPS TRAILING BALNKS
PSORT			ACRPOSR	ACRDICT	RESOLVES THE DATA NAMES AS SOURCE OR TARGET
RSPACE	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RSRCTAR	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RSUBPTS		1-PTR 2-PTR	RETREVE RETRPE	MERGSUB	RETURNS IN THE SECOND ARGUMENT, A POINTER TO THE SUBSCRIPT RANGE FOR THE SUBSCRIPT NAME REPRESENTED BY THE FIRST POINTER.
RSUBIT	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RTAG	RPAGE	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
RTLABEL	RPAGE	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
PTRIMR	CONV	1-CHAR(*) *-CHAR(100)VAR			SKIPS TRAILING BLANKS.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
[illegible]	[illegible]	*-BIT(1)		RMEDIAT	USED TO RECOGNIZE SAP KEYWORDS.
[illegible]	[illegible]	*-BIT(1)			USED TO RECOGNIZE SAP KEYWORDS.
[illegible]		1-PTR 2-PTR 3-PTR		ACRDD UPDASSN	UPDATES THE DT OF AN ASSERTION TO INCLUDE UPDATED SUBSCRIPTS IN THE DATA NAMES USED IN THE ASSERTION.
SCOMADD	SCOMINT	1-CHAR(*)VAR		LEX	ADD A COMMENT STRING TO STACK.
SCOMINT				MONITOR	USED TO INITIALIZE COMMENT STACK (SYNTAX ANALYSIS).
SCONSET	SCOMINT			ACRDICT GENASSI GENSTMT SSTORE STFLD SVSTV UPDICT	INITIALIZE DATA AREA OF A STORAGE ENTRY.
[illegible]	[illegible]	1-PTR 2-BIN FIXED		ASUBST MWARN1	MOVES AN ERROR CODE (GIVEN BY THE 2ND ARGUMENT) INTO THE DATA AREA OF THE STORAGE ENTRY REPRESENTED BY THE FIRST ARGUMENT.
SENTRY	[illegible]	1-BIN FIXED 2-BIN FIXED *-BIT(1)		ODTAP REMOVEG	RETURNS 1 IF THE DICTIONARY ENTRIES REPRESENTED BY THE TWO ARGUMENTS ARE SAME.
[illegible]			ACRDFP CONVPTR	RESIC	OBTAINS THE NUMBER OF DICTIONARY ENTRY FOR EACH DATA NAME.
SETDTMP		1-PTR 2-PTR	ACRDD	ACRCHST ACRDUP ASUBST CSUBLST DUPLDSK IFENAMS RESIC	ESTABLISHES PARENT DAUGHTER RELATIONSHIPS
SETPR		1-PTR	ACRDR ADJMSET GDINDEX SETTYPE		SETS PRECEDENCE RELATIONS BETWEEN THE SOURCE VARIABLES, ASSERTION (SPECIFIED BY THE ARGUMENT DICTIONRY ENTRY) AND THE TARGET VARS OF THE ASSERTION.
[illegible]		1-BIN FIXED 2-BIN FIXED	ACRDNF ACRDR GENSUB		SUPPLIES A SUBSCRIPT NAME FOR THE J TH DIMENSION OF DICT. ENTRY I (WHERE I AND J ARE THE TWO ARGUMENTS).
SETSUBR		1-PTR	RETREVE RETRFE	ACRSTVS	SETS THE RANGE OF A SUBSCRIPT NAME FROM THE RANGE SUPPLIED BY A FOR EXPRESSION.
SETTYPE		1-BIN FIXED	ACRPOSR ADJMSET CHKCONT	RICOMPE SETPR UPDICT	ESTABLISHES TYPE-1, TYPE-2, AND TYPE-8 RELATIONSHIPS

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SETTP5		1-PTR 2-BIN FIXED	ADJMSET RETREVE RETRPE	ACRADJM	ESTABLISHES TYPE-5 AND TYPE-6 RELATIONSHIPS
SETTP9			ACRDR ADJP ADJMSET DATATYPE GDINDEX GDTKPTR	RESIC	ESTABLISHES TYPE-9 RELATIONS.
SGETNAM	SINITSN	1-CHAR(*)		SSTORE	RETURNS A NAME IN STACK.
SGEXIST	SINITSN	1-CHAR(*)VAR		STFLD	RETURNS THE EXIST NAMES IN THE STACK.
[illegible]	SINITSN			SSTORE STFLD	RETURNS A NAME IN STACK.
[illegible]	SINITSN			IFIELD SVANAM	INCREMENT # OF NAMES IN STACK.
[illegible]	SINITSN				INCREMENT KEY NAME.
[illegible]	SINITSN			RMEDIAT	INCREMENT KEY NAME.
SINITSN			ERSYST		SAVES DATA NAMES IN A STACK DURING SYNTAX ANALYSIS.
SINTKEY	SINITSN	4-BIT(1)		XVUNIT	RETURNS 1 IF KEY IS SPECIFIED FOR A DD STATEMENT.
SORTASS		1-(*)PTR 2-BIN FIXED	ACRDP ACRDRA	PRTASSX	SORTS ASSERTION NAMES ALPHABETICALLY.
[illegible]		1-CHAR(*)VAR 2-PTR 3-BIN FIXED 4-BIT(1)		ACNPTR	RETURNS 1 IF THE NAME (FIRST ARG) IS CONTAINED IN THE FILE STRUCTURE(2ND ARG)
SSTMOD	SSTORE				STORES MODULE STATEMENT IN AM.
SSTORE		1-CHAR(*)	ERSSEM SCOMSET SGETNAM SGGNAME STORE ERGTN1		USED TO STORE A DD STATEMENT
SSTORE1	SSTORE	1-CHAR(2)			USED TO STORE A DD STATEMENT.
SSTREF	SSTORE				SAVES REFER STATEMENT IN AM.
SSTSRC	SSTORE				SAVES SOURCE AND TARGET FILE STATEMENTS IN AM.
SSUBPTR		1-PTR 2-PTR		MERGSUB	UPDATES A FOR-CLAUSE TO CONTAIN APPROPRIATE SUBSCRIPT RANGE.

PROC. OR ENTRY PT.	INTERFC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SSUBRNG		1-PTR	ERSYST GDINDEX GPPTR	CSUBLST IFENAMS	SETS RANGE OF A SUBSCRIPT.
STFILE	STFLD				USED TO STORE A FILE STATEMENT IN THE ASSOCIATIVE MEMORY.
STFLD			ERSYST GPICPTR IUSILEL SCOMSLT SGGNAME STORE DATAAME RECLPUF SGEXIST		STORES FIELD, GROUP, RECORD, SUBSCRIPT STATEMENTS IN THE ASSOCIATIVE MEMORY.
STMEDIA	STORE				STORES A MEDIA STATEMENT IN AM.
STMTDEL		1-PTR 2-BIN FIXED		CSUBL CSUBRED	DELETES A STORAGE ENTRY TEMPORARILY, BY MOVING AN ERROR CODE TO THE DATA AREA.
STMTNO		1-PTR *-CHAR(4)		WARNT	RETURNS THE STATEMENT NO OF THE ARGUMENT STORAGE ENTRY.
STORE		1-CHAR(*) VAR 2-PTR	CONVPTR	ACRDICT DUPLSE GENASSI GENSTAT SSTORE STFLD SVANAP SVSTV UPDICT	STORES A STATEMENT IN ASSOCIATIVE MEMORY
SUBPRC		1-BIN FIXED 2-BIN FIXED *-BIT(1)	ACRDR CSUBPT REMDUPL NOOTFIP UNIONL	SUBPROP	RETURNS '1'B IF THE SUBSCRIPT NAMES REPRESENTED BY THE TWO ARGUMENT DICTIONARY ENTRIES ARE SAME.
SUBPROP			ACRDF ACRDNA ACFFCSR DATATYPE FREEL GDINDEX GDTRPTR GEGNAME GVARDI LISTSA LISTSI LISTSL NODTYPE REMDUPL SUBTRL UNIONL WERROR XNLIST SUBPRC SUBPRU UPDINTM CONVT CPLINE# LISTGET NODTYPN PRTWPM1	MONITOR	PERFORMS SUBSCRIPT PROPAGATION.
SUBSCRT		1-PTR *-BIT(1)	GENSUB RECLBUF RETREVE WTRAE RETRNAM	ANALIND GET_AE	RETURNS 1 IF THE ARGUMENT NAME IS A SUBSCRIPT NAME.
SUBTRL	XNLIST	1-PTR 2-PTR 3-PTR		SUBPROP	USED TO SUBTRACT A LIST FROM ANOTHER.
SUBPCHF			ACRDFP ACRDP DATATYPE MERGSUB	NFSIC	SUPPLIES FOR EACH EXPRESSION FOR THE ASSERTIONS THAT USE SUBSCRIPTED DATA NAMES.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SWAAF1	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF2	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF3	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF5	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF6	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF7	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAF8	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAS2	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAS3	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAS4	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAAS6	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWACFT1	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWACFT2	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAFCS1	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAFCS2	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWAF9	SWANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SVAP2	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVANAR			CONVPTR DUMPHEX LEX LEXBUFN LEXCCET LEXCLEN LEXTMPG RFLDTMP SIMCRQN STORE SVSTV		USED TO SAVE THE DERIVATION TREE OF AN ASSERTION.
SVANP1	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVANP2	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVADP1	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVADP3	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP2	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP3	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP31	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP4	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP5	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP51	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP6	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP7	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAP8	SVANAR				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SVAPFL1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAPFL2	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAPFL3	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAPFL4	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVASPE1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVASPE2	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVASPE3	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVASS1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVASS2	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVATS1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVATS2	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVATS3	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVATS4	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAT1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAT2	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SVAV1	SVANAP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
SWAJT	SWDREP				USED IN SAVING THE DERIVATION TREE OF AN ASSERTION.
SWREF			LEXSREF LIBREFS		USED TO SAVE REFER STATEMENTS.
SWSHELP	SWHELP				PROCESSES HELP STATEMENTS.
SWSHELP	SWHELP				PROCESSES HELP STATEMENTS.
[illegible]	JUNK				
[illegible]	JUNK				
[illegible]	JUNK				
SWSTV			RETREVE SCOPSET STORE NWARN	SVANAM	USED TO INITIALIZE THE DATA AREA OF AN ASSERTION.
SWSTV1	SWSTV				USED TO INITIALIZE THE DATA AREA OF AN ASSERTION.
SWHELP					PROCESSES HELP STATEMENTS.
TESTFUN		1-PTR 2-BIT(1)	RECLFUF	ACRCHST	USED TO TEST IF THE ARGUMENT DATA NAME IS A FUNCTION NAME.
TESTMED		1-BIN FIXED 2-BIT(1)	ACPDR	GRTAP REMOVES	RETURNS 1 IF THE ARGUMENT DICT. ENTRY REPRESENTS A MEDIA NAME.
TESTRP	PPASE	1-BIT(1)			USED TO RECOGNIZE RIGHT PARANTHESIS IN SAP.
TANPRNT			LEX LEXDSAB LEXENAS RECNAME RKWORDF RKWORDL		USED TO OBTAIN PARENT NAME OF A DATA NAME.
TRACE		1-CHAR(*) 2-PTR	CONVPTR DUPPHEX PRTHEXC		USED TO OBTAIN PROGRAM TRACE.
TRACEN	TRACE	1-BIN FIXED 2-CHAR(*) 3-PTR			USED TO OBTAIN PROGRAM TRACE.
TRACES	TRACE				
TRACEX	TRACE	1-BIN FIXED 2-CHAR(*) 3-PTR 4-BIN FIXED			USED TO OBTAIN PROGRAM TRACE.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
TRACE1	TRACE				
UNIQSE		1-PTR 2-PTR			RETURNS THE UNIQUE STORAGE ENTRY PTR FOR THE ARGUMENT SE PTR.
UNION	ANLIST	1-PTR 2-PTR 3-PTR 4--BIN FIXED		SUBPRC SUBPROP	USED TO OBTAIN UNION OF TWO ARGUMENT LISTS.
UPDASGN		1-BIN FIXED 2-PTR 3-BIN FIXED	ACRDR ACRDRA ACRPOSR ERSYST RWINDT	IFENAMS	UPDATES A VARIABLE IN AN ASSERTION.
UPDDIM		1-PTR 2-BIN FIXED	ACRDRSV DATAYPE ERSYST WARNS	ASUBST	UPDATES THE DIMENSION OF A DATA NAME.
UPDDIPT	UPDDIM	1-PTR		UPVLIST	UPDATES THE DIMENSION OF A DATA NAME.
UPDDTR		1-PTR	RETRIEVE RETRWE UPDDTRD	ACRDICT	SUPPLIES DAUGHTER NAMES FOR THOSE GROUP OR RECORD NAMES FOR WHICH NO DESCENDENTS EXIST.
UPDDTRD		1-PTR 2-PTR 3-PTR	CONVPTR DUPLSE	UPDDTR	UPDATES THE DAUGHTER LIST OF AN ENTRY.
UPDICT		1-CHAR(*)VAR 2-PTR	ACRDR ADJMSET GPINDEX IUSERA SCOMSET SETTYPE STORE	GAFSUBS	UPDATES AR AND DICTIONARY
UPDIMT		1-BIN FIXED 2-PTR		SUBPROP	UPDATES THE DIMENSION OF AN INTERIM DATA NAME
UPVLIST		1-BIN FIXED	ACRDR ACRDRSV UPDDIMT	GRNGDIM	UPDATES THE VARIABLE LIST STRUCTURE OF A DICTIONARY ENTRY.
WACCT	WUFEC				ERROR STACKING ROUTINE USED IN SAP.
WUDAFS	WUFEC				ERROR STACKING ROUTINE USED IN SAP.
WUDS	WUFEC				ERROR STACKING ROUTINE USED IN SAP.
WUSN1	WUFEC				ERROR STACKING ROUTINE USED IN SAP.
WUSN2	WUFEC				ERROR STACKING ROUTINE USED IN SAP.

THIS PAGE IS BEST QUALITY PRACTICABLE
FROM COPY FURNISHED TO DDC

PROC. ON ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
WLIGHTV		1-(3)BIN FIXED 2-(3)BIN FIXED 3-BIN FIXED 4-BIN FIXED 5-(5)BIN FIXED		CHKCONT LOOPAV MAPV	RETURNS MAP VECTOR.
WLSUPL	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
WERROR		1-CHAR(*)VAR		CHKSCON SUBPROP WWARNT	WRITES SEMANTIC ERROR MESSAGES TO ERROR FILE.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]STAG			RETRIEVE WWARNIT WWARNGT		WRITES ALL WARNING MESSAGES.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]OUT		1-CHAR(*)		LIBADDR LIBU PRTASSA	WRITES AN ERROR MESSAGE TO SYSPRINT, AND THEN STOPS.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
WSAPCH			WSAPOUT	ACRDICT	WRITES A HEADING TO SAPOUT FILE.
[illegible]	WSAPCH				WRITES A HEADING TO SAPOUT FILE.
[illegible]	WSAPCH				WRITES A HEADING TO SAPOUT FILE.
WSAPOUT		1-CHAR(*)VAR		GENSTMT LEXREAD MONITOR WSAPCH	WRITES A MESSAGE TO SAPOUT FILE.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
[illegible]	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
WSIZEL	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
WSIZEO	WSPEC				ERROR STACKING ROUTINE USED IN SAP.
WSPACE	WSPEC				ERROR STACKING ROUTINE USED IN SAP.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
WTA.	[illegible]				ERROR STACKING ROUTINE USED IN SAP.
[illegible]			SPUSH		
[illegible]	[illegible]				ERROR STACKING ROUTINE USED IN SAP.
[illegible]	[illegible]				ERROR STACKING ROUTINE USED IN SAP.
WWARN				RTCONSE SYSTV WWARNP WWARN1	USED TO WRITE A WARNING MESSAGE.
WWARNP		1-BIN FIXED 2-PTR 3-CHAR(*)VAR	DATAAME WWARN CONVR		WRITES A MESSAGE TO WARNING FILE.
WWARN1			CONVF DATAAME DATATYPE GAMMFV LISTGET LISTSL PATASS1 PHTSTAT STMTNO WWARN LISTSA PALTSNM SEVERR WERROR	AERDICT	USED TO WRITE A WARNING MESSAGE.
WWARN11	WWARN1			WMESSAG	WRITES A MESSAGE TO WARNING FILE.
[illegible]	WWARN			AERDICT	USED TO WRITE A WARNING MESSAGE.
[illegible]	WWARN			GENASSI	USED TO WRITE A WARNING MESSAGE.
[illegible]	WWARN			GENASSI GENSTMT	USED TO WRITE A WARNING MESSAGE.
[illegible]	WWARN1			WMESSAG	WRITES A MESSAGE TO WARNING FILE.
[illegible]	WWARN			GFUDIP	USED TO WRITE A WARNING MESSAGE.
XAREF		1-CHAR(*)VAR	CONVF CONVRF	XREFPRT XWRITE1 XWXREF	WRITES AN OUTPUT LINE TO XREF FILE.
XAREFA	XAREF	1-BIN FIXED 2-BIT FIXED(31) 3-BIN FIXED 4-CHAR(*)VAR 5-CHAR(*)VAR		XWRITE1	WRITES AN OUTPUT LINE TO XREF FILE.
XNLIST		1-PTR 2-PTR 3-PTR	ENSUF	SUBPROP	OBTAINS INTERSECTION OF THE TWO ARGUMENT LISTS. THE RESULT IS RETURNED IN THE THIRD ARGUMENT.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
XXREFRPT			XXREF XWRITLT	MONITOR	PRINTS CROSS REFERENCE REPORT.
XWBLK	XWUNIT	*--BIT(1)		RMEDIAT RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWCHANC	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWDEN	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWDISP	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWDSN	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWELABEL	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWILABEL	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWKEY	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWLABEL	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWLINE	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWLNGTH	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWSPC	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWPAGE	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWPARTY	XWUNIT	*--BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.

PROC. OR ENTRY PT.	IN PROC.	ARGUMENTS	CALLS	CALLED BY	DESCRIPTION
XVREC	XVUNIT	*-BIT(1)		RMEDIAT RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XVRECFM	XVUNIT	*-BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XVSUNIT	XVUNIT	*-BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XVTAP	XVUNIT	*-BIT(1)		RPAGE	USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XVUNIT			SINTKEY	RPACE	
XVXFILE	XVUNIT	*-BIT(1)			USED TO CHECK THE PRESENCE OF A KEYWORD IN A DD STATEMENT.
XWGATTR		1-PTR	LISTSA DATATTR RECLBUF XWSIZE	XWRITE1	OBTAINS THE ATTRIBUTES OS A STORAGE ENTRY.
XWRITE1		1-PTR	LISTGET LISTSI LISTSL RECLBUF WXREF WXRLFA XWGATTR XWXREF	XREFPRT	USED TO GENERATE XREF REPORT
XWSIZE		1-PTR	CONVRT LISTSA RECLBUF	XWGATTR	USED TO GENERATE XREF REPORT
XWXREF		1-PTR	ACRDR ACRPOSR CONVR LISTGET LISTSA LISTSI LISTSL WXREF	XWRITE1	USED TO GENERATE XREF REPORT

Appendix B

This appendix lists the information displayed by each use of the HELP command. The usage of each command is denoted by the entries in lower case letters.

```
 help model

FOLLOWING TYPES OF STATEMENTS ARE ACCEPTED BY MODEL:
   (1).  MODULE STATEMENT (MODULE).
   (2).  SOURCE FILE STATEMENT (SOURCEF).
   (3).  TARGET FILE STATEMENT (TARGETF).
   (4).  REFER STATEMENT (REFER).
   (5).  MEDIA DESCRIPTION STATEMENT (MEDIA).
   (6).  FILE DESCRIPTION STATEMENT (FILE).
   (7).  RECORD DESCRIPTION STATEMENT (RECORD).
   (8).  GROUP DESCRIPTION STATEMENT (GROUP).
   (9).  FIELD DESCRIPTION STATEMENT (FIELD).
   (10).  INTERIM DESCRIPTION STATEMENT (INTERIM).
   (11).  SUBSCRIPT DESCRIPTION STATEMENT (SUBSCRIPT).
   (12).  ASSERTIONS (ASSERTION).

   TO DISPLAY THE SYNTAX AND DESCRIPTION OF ANY STATEMENT ENTER:
     HELP <STMT-TYPE> WHERE <STMT-TYPE> IS THE STATEMENT TYPE, WHICH
IS ENCLOSED IN PARENTHESES IN THE ABOVE LIST.
   FOR EXAMPLE, THE FOLLOWING STATEMENT:
     HELP FIELD WILL DESCRIBE THE SYNTAX AND THE DESCRIPTION OF
FIELD STATEMENT.

 help module

PURPOSE: TO GIVE A NAME TO THE DESIRED MODULE.  SYNTAX:
    <MODULE-STATEMENT>::= <NAME> IS MODULE;

 help sourcef

PURPOSE: TO INDICATE THE NAMES OF THOSE FILES WHICH ARE SOURCE OR
INPUT TO THE DESIRED MODULE.  SYNTAX:
    <SOURCE-FILE-STATEMENT>::= SOURCE [ FILE ] :
                 <FILENAME> [, <FILENAME> ]* WHERE <FILENAME> IS THE
NAME OF A FILE.

 help targetf

PURPOSE: TO INDICATE THE NAME OF THOSE FILES WHICH ARE TARGET OR
OUTPUT TO THE DESIRED MODULE.  SYNTAX:
    <TARGET-FILE-STATEMENT>::= TARGET [ FILE ] :
```

<FILENAME> [, <FILENAME>]* WHERE <FILENAME> IS THE NAME OF A FILE.

help refer

PURPOSE: TO INCLUDE A SPECIFIC SECTION OF A MODULE IN THE LIBRARY.
SYNTAX:
```
  <REFER-STATEMENT>::= REFER : <MODULE-NAME> <SECTION-NAME>
```

WHERE- <SECTION-NAME> IS THE NAME OF THE SECTION (IN THE MODULE SPECIFIED BY <MODULE-NAME>) WHICH IS TO BE INCLUDED IN THE MODEL SPECIFICATION.

help help

"HELP" IS A TEACHING AID PROVIDED IN MODEL. IT DESCRIBES THE ALLOWED STATEMENTS IN MODEL AND THE CORRESPONDING SYNTAX. IT ALSO GIVES A BRIEF DESCRIPTION OF THE ERROR CODES DISPLAYED DURING SYNTAX ANALYSIS. THERE ARE ESSENTIALLY FOUR USES OF HELP:

(1). HELP MODEL

THIS USE DISPLAYS ALL THE ALLOWED STATEMENTS IN MODEL.

(2). HELP SYNTAX

THIS USE DISPLAYS THE CONVENTIONS USED IN THE SYNTAX OF MODEL STATEMENTS.

(3). HELP <STATEMENT-TYPE>

THIS USE DISPLAYS THE SYNTAX AND DESCRIPTION OF A PARTICULAR TYPE OF STATEMENT SPECIFIED BY <STATEMENT-TYPE>.

(4). HELP <ERROR-CODE>

THIS USE DISPLAYS A SHORT DESCRIPTION OF A PARTICULAR TYPE OF ERROR SPECIFIED BY <ERROR-CODE>.

IF NO ARGUMENT IS SPECIFIED IN THE "HELP" STATEMENT, THEN A SHORT DESCRIPTION OF THE MOST RECENT SYTAX ERROR ENCOUNTERED DURING THE SYNTAX ANALYSIS IS DISPLAYED. IF NO ERROR WAS ENCOUNTERED, THEN "HELP MODEL" IS ASSUMED.

help syntax

IN THE DESCRIPTION OF THE SYNTAX OF MODEL STATEMENTS, THE FOLLOWING CONVENTIONS WILL BE USED:

```
   (1).  ITEMS INSIDE SQUARE BRACKETS ARE OPTIONAL.
   (2).  IF AN ASTERISK FOLLOWS A RIGHT SQUARE BRACKET,
        THEN THE ITEM INSIDE THE SQUARE BRACKET CAN REPEAT
        ZERO OR MORE TIMES.
   (3).  { X | Y } - INDICATES THE SELECTION OF EITHER X OR Y.
   (4).  <X> REFERES TO A GENERIC CLASS OF ITEMS FOR WHICH A SPECIFIC
        ITEM NEEDS TO BE SUBSTITUTED.
   (5).  ALL OTHER WORDS REFER TO SPECIFIC MODEL VOCABULARY.

  BUT FOR  THE CONVENTIONS (1), (2)  AND (3), THE SYNTAX  USES REGULAR
  BNF NOTATION.


   help media


  PURPOSE: DESCRIBES THE MEDIUM IN WHICH THE SOURCE OR TARGET
           FILE RESIDES.  SYNTAX:
   <MEDIA-STATEMENT>::= <NAME> IS MEDIA
                         [(<ARGUMENT>[,<ARGUMENT>]*)]      WHERE-       THE
  <ARGUMENT> HAS THE FOLLOWING SYNTAX:
   <ARGUMENT>::= BLOCKSIZE = <INTEGER>
               | RECORDSIZE = <INTEGER>
               | ORG = {SAM | ISAM | DIRECT}
               | RECFM = {FIXED | FB | VARIABLE |
                      VB | VS | UNDEFINED}
               | UNIT = { TAPE | CARD | PUNCH | DISK |
                     TERMINAL }
               | DISP = { OLD | SHR | NEW | MOD }
               | TLABEL = { SL | NL | BLP }
               | PARITY = { ODD | EVEN }
               | CODE = { EBCDIC | BCD | ASCII }
               | TRACKS = { 7 | 9 }
               | DENSITY = { 200 | 556 | 800 | 1600 }
               | IL = <NAME>
               | EL = <NAME>
               | TAB = (<INTEGER> [,<INTEGER>]*)
               | PAGESIZE = <INTEGER>
               | LINESIZE = <INTEGER>

   help file

  PURPOSE: TO DESCRIBE A FILE AND ITS ATTRIBUTES.  SYNTAX:
   <FILE-STATEMENT>::= <NAME> IS FILE [(<PARENT-NAME>
                        [,<ARGUMENT>]*)] WHERE- THE <ARGUMENT> HAS THE
  FOLLOWING SYNTAX:
   <ARGUMENT>::= BLOCKSIZE = <INTEGER>
               | RECORDSIZE = <INTEGER>
               | ORG = {SAM | ISAM | DIRECT}
               | RECFM = {FIXED | FB | VARIABLE |
                      VB | VS | UNDEFINED}
               | UNIT = { TAPE | CARD | PUNCH | DISK |
                 TERMINAL }
           | DISP = { OLD | SHR | NEW | MOD }
           | TLABEL = { SL | NL | BLP }
           | PARITY = { ODD | EVEN }
           | CODE = { EBCDIC | BCD | ASCII }
           | TRACKS = { 7 | 9 }
           | DENSITY = { 200 | 556 | 800 | 1600 }
           | IL = <NAME>
           | EL = <NAME>
           | TAB = (<INTEGER> [,<INTEGER>]*)
           | PAGESIZE = <INTEGER>
           | LINESIZE = <INTEGER>
           | KEY = <KEY-NAME>
           | DSN = <DATASET-NAME>
           | FILE# = <INTEGER>
           | SPACE = ({TRK | CYL | <INTEGER>}
                    [,[<PRIMARY>][,[<SECONDARY>]
                    [,[<DIRECTORY>]]]
```

WHERE <PRIMARY>, <SECONDARY> AND <DIRECTORY> ARE INTEGER CONSTANTS THAT SPECIFY THE UNITS REQUIRED FOR PRIMARY EXTENT, SECONDARY EXTENT AND THE DIRECTORY OF THE DATASET RESPECTIVELY. ALSO, THE <PARENT-NAME> IN THE ABOVE SYNTAX SPECIFIES THE PARENT MEDIA NAME.

help record

PURPOSE: TO DESCRIBE A RECORD OF A FILE AND ITS ATTRIBUTES. SYNTAX:
<RECORD-STATEMENT>::= <NAME> IS RECORD
[(<PARENT-NAME>[,<ARGUMENT>]*)] WHERE- <ARGUMENT> HAS THE FOLLOWING SYNTAX:
<ARGUMENT>::= KEY = <KEY-NAME>
| (<SIZE> [,<SIZE>]*) WHERE- <KEY-NAME> IS THE NAME OF THE KEY FIELD AND <SIZE> IS A
NAME OR INTEGER CONSTANT THAT SPECIFIES THE SIZE OF A DIMENSION OF THE RECORD.

help group

PURPOSE: TO DESCRIBE A GROUP IN A FILE OR IN A RECORD OF A FILE.
SYNTAX:
<GROUP-STATEMENT>::= <NAME> IS GROUP
[(<PARENT-NAME>[,<ARGUMENT>]*)] WHERE- <ARGUMENT> HAS THE FOLLOWING SYNTAX:
<ARGUMENT>::= KEY = <KEY-NAME>
| (<SIZE> [,<SIZE>]*) WHERE- <KEY-NAME> IS THE NAME OF THE KEY FIELD AND <SIZE> IS A
NAME OR INTEGER CONSTANT THAT SPECIFIES THE SIZE OF A DIMENSION OF THE GROUP.

help field

```
PURPOSE: TO DESCRIBE A FIELD IN A GROUP OR A RECORD STRUCTURE OF
        A FILE.  SYNTAX:
 <FIELD-STATEMENT>::= <NAME> IS FIELD
           [(<PARENT-NAME>[,<ARGUMENT>]*)]  WHERE-  <ARGUMENT>  HAS
THE FOLLOWING SYNTAX:
 <ARGUMENT>::= (<SIZE> [,<SIZE>]*)
             | CHAR [(<LENGTH>)] [VAR]
             | BIT [(<LENGTH>)] [VAR]
             | BIN {FIXED | FLOAT} [(<PRECISION>)]
             | DEC {FIXED | FLOAT} [(<PRECISION>)]
             | PIC ['<PICTURE-SPECIFICATION>'] WHERE-
   <SIZE> AND  <LENGTH> ARE EITHER  INTEGER CONSTANTS OR  NAMES THAT
RESPECTIVELY  SPECIFY  THE  SIZE  OR  LENGTH  OF  THE  FIELD  NAME.
<PICTURE-SPECIFICATION>  SPECIFIES  THE PICTURE  ATTRIBUTES  OF  THE
FIELD NAME AND THE <PRECISION> FOLLOWS THE FOLLOWING SYNTAX:
   <PRECISION>::=    <INTEGER>[,<INTEGER>]   AND  THE   TWO   INTEGER
CONSTANTS SPECIFY THE NUMBER OF DIGITS  AND THE NUMBER OF FRACTIONAL
DIGITS FOR THE DATA NAME.
```

help interim

```
PURPOSE: TO DESCRIBE THE ATTRIBUTES OF THE INTERIM DATA
        NAMES USED IN THE SPECIFICATION.  SYNTAX:
 <INTERIM-STATEMENT>::= <NAME> IS INTERIM
                   [(<ARGUMENT>[,<ARGUMENT>]*)] WHERE-
 <ARGUMENT>::= (<SIZE> [,<SIZE>]*)
             | CHAR [(<LENGTH>)] [VAR]
             | BIT [(<LENGTH>)] [VAR]
             | BIN {FIXED | FLOAT} [(<PRECISION>)]
             | DEC {FIXED | FLOAT} [(<PRECISION>)]
             | PIC ['<PICTURE-SPECIFICATION>'] WHERE-
   <SIZE> AND  <LENGTH> ARE EITHER  INTEGER CONSTANTS OR  NAMES THAT
RESPECTIVELY  SPECIFY  THE  SIZE  OR LENGTH  OF  THE  INTERIM  NAME.
<PICTURE-SPECIFICATION>  SPECIFIES  THE PICTURE  ATTRIBUTES  OF  THE
INTERIM NAME AND THE <PRECISION> FOLLOWS THE FOLLOWING SYNTAX:
   <PRECISION>::=    <INTEGER>[,<INTEGER>]   AND  THE   TWO   INTEGER
CONSTANTS SPECIFY THE NUMBER OF DIGITS  AND THE NUMBER OF FRACTIONAL
DIGITS FOR THE DATA NAME.
```

help subscript

```
PURPOSE: TO DESCRIBE THE ATTRIBUTES OF A SUBSCRIPT NAME.  SYNTAX:
 <SUBSCRIPT-STATEMENT>::= <NAME> IS SUBSCRIPT
         [(<PARENT-NAME>[,[<LOWER-BOUND>]
         [,[<UPPER-BOUND>][,<INCREMENT>]]])] WHERE-
      <LOWER-BOUND>, <UPPER-BOUND> AND <INCREMENT> ARE
      INTEGER CONSTANTS.
```

help assertion

```
PURPOSE: TO SPECIFY THE RELATIONSHIPS BETWEEN DATA NAMES
         USED IN THE MODEL SPECIFICATION.  SYNTAX:
 <ASSERTION-STATEMENT>::= [<ASSERTION-NAME> :] <ASSERTION>
 <ASSERTION>::= <IF-CLAUSE> <ASSERTION>
                             [ELSE <ASSERTION>]
          | <FOR-CLAUSE> <ASSERTION>
          | <DATA-NAME> = <ARITHMETIC-EXPRESSION> WHERE-
 <IF-CLAUSE>::= IF <BOOLEAN-EXPRESSION> THEN
 <FOR-CLAUSE>::= FOR <SUBSCRIPT-NAME> = <LOWER-BOUND>
                [ TO <UPPER-BOUND>] [BY <INCREMENT>] LET WHERE-
   <LOWER-BOUND>, <UPPER-BOUND>, AND <INTEGER> ARE INTEGER
   CONSTANTS; AND <BOOLEAN-EXPRESSION> AND
   <ARITHMETIC-EXPRESSION> HAS THE REGULAR SYNTAX (AS IN PL/1).
```

INDEX

BIBLIOGRAPHY

ASH77 Ashcroft, E.A., and W. W. Wadge, "LUCID: A Programming Language with Iteration", Communications of ACM, Vol. 20, No. 7, July 77, pp 519-526.

BAL72 Balzer, R.M., "Automatic Programming", Memorandum, USC Information Science Institute, Sept. 1972.

BAL74 Balzer, R., et. al, "Domain Independent Automatic Programming", USC/Information Sciences Institute RR-77-14, 1974.

BIE76 Biermann, A.W., "Approaches to Automatic Programming", in Advances in Computers, Ed. M. Rubinoff, M.C. Yovits, Volume 15, pp 1-63, 1976.

BOS76 Bosyj., M., " A Program for the Design of Procurement Systems", Tech. Rep. 160, Laboratory for Computer Science, MIT, 1976.

CHA77 Chang, Y., "Automatic Test Program Generation", Ph.D. Thesis in Computer and Information Sciences, University of Pennsylvania, 1977.

CHE76 Che, H., and Y. Chang, " The NOPAL Language, Specification and User Manual", Moore School Report No. 76-04, University of Pennsylvania, 1976.

CHE78 Che, Her-daw, "Exploit Parallesim in Non-procedural

Programs", Dissertation Proposal, Moore School of Electrical Engg., University of Pennsylvania, 1978.

COU73 Cougar, J.D., "Evolution of Business System Analysis Techniques", Computing Surveys, Vol. 5, No. 3, Sept. 73.

EAR74 Earley, J., "High Level Operations in Automatic Programming", Sigplan Notices, Proc. of Symposium on Very High Level Languages, Vol. 9, No. 4, Apr 74, pp. 34-42.

FRE72 French, A., "A Syntax Analysis Program Generator", Masters Thesis, Computer and Information Sciences Department, Moore School of Electrical Engineering, University of Pennsylvania, 1972.

GAN78 "Use and Extensions of an Automatic Program Generator System for Model building in the Social and Engineering Sciences", Ph.D. Dissertation, Dept. of Computer and Information Sciences, University of Pennsylvania, 1978.

GIB75 Gibb, K.R., "Automatic File and Module Design", Thesis Proposal, Dept. of Computer and Information Science, University of Penna., 1975.

GOL75A Goldberg, P.C., "Automatic Programming", In Programming Methodology, Lecture Notes in Computer Science, Ed. G. Goos, & J. Hartmanis, Springer-Verlag, 1975.

GOL75B Goldberg, P.C., "Structural Programming for Non-programmers", RC-5318, IBM Yorktown Heights, March 75.

GRE76 Green, C., "The Design of PSI Program Synthesis System", Second International Conf. on Software Engg., San Francisco, California, Oct. 76, pp 4-18

GRI54 Griffin, H., "Elementary Theory of Numbers", McGraw Hill Co., Inc. 1954.

HAM74 Hammer, M.M., W. G. Howe, I. Wladawsky, "An Interactive Business Definition System", Sigplan Notices, Proc of Symposium on Very High Level Languages, Vol. 9, No. 4, Apr. 74, pp. 25-33.

HAM77 Hammer, M., W.G. Howe, V.J. Kruskl & I. Wladawsky, "A Very High Level Programming Language for Data Processing Applications", Communications of ACM, Vol. 20, No. 11, Nov. 77, pp 832-840

HEI76 Heidorn, C.E., "Automatic Programming Through Natural Language Dialogue: A Survey", IBM J of R and D, Vol. 20, No. 4, July 76, pp 302-313.

LEA74 Leavenworth, B.M., and J.E. Sammet, "An Overview of Non-procedural Languages", Proc. Symposium on Very High Level Languages, Sigplan Notices, Vol. 9, No. 4, 1974, pp 1-12.

LEA77 Leavenworth, "Non-procedural Data Processing", Computer Journal, Vol. 20, No. 1, Jan. 77, pp 6-9.

LIE77 Lieberman, L.I., & M.A. Wesley, "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly", IBM J of R & D, Vol. 21, No. 4, July 77.

LON77 Long, W. J., "A Program Writer", Ph.D. Dissertation, MIT, August 1977.

McC70 McCracken, D., and U. Garbassi, " A guide to COBOL Programming", 2nd Edition, Wiley - Interscience, 1970.

MAL75 Malhotra, A., "Design Criteria for Knowledge Based English Language System for Management: An Experimental Analysis", MAC TR-146, Project MAC, MIT 1975.

MAR74 Martin, W. A., "OWL: A System for Building Expert Problem Solving Systems Involving Verbal Reasoning", Notes from Course 6871, MIT, Cambridge, 1974.

MIC68 D. Michie, "Machine Intelligence 3," American Elsevier, NY 1968.

ODO77 M. J. O'Donnel, "Computing in systems Described by Equations," Lecture notes in Computer Science, G.

Goos & J. Hartmanis, Eds., Springer-Verlag, 1977

PET76 Peterson, N.D., "COBOL Generation of Source Programs", Software - Practice & Experience, VOL.6, 1976, pp 117-131

PET77 Peter de Jong, S., and M.M. Zloof, "Communication with the System for Business Automation (SBA), RC-6788, IBM T.J. Watson Research Center, 1977.

PNU76A Pnueli, A., "A Simple Specification Language to ADP and its Implementation", Working Paper, APG Project, Moore School, Univ. of Penna., 1976.

PNU77 Pnueli, A., and N.S. Prywes, "Compiling Iterations in Non-procedural Languages", Working Paper, Moore School, University of Pennsylvania, 1977.

PRY74 Prywes, N.S., "Automatic Generation of Software Systems", Data Base, Vol. 6, No. 2, Summer 1974, pp 7-16.

PRY75A Prywes, N.S., "Automatic Computer Program Generation for Automatic Testing Systems", Rep. FCF-3-75, U.S. Army, Frankford Arsenal, Philadelphia, Pennsylvania, 1975.

PRY76A Prywes, N.S., "Automatic Generation of Computer Programs for Converting Transmitter Data to IRS tape Standards", Phase-I Report, Contract# TIR-7T-62, IRS Planning & Research Division, Wash.

D.C., 1976.

PRY76B Prywes, N.S., "Concept of English to MODEL Interface (EMI) for Specifying Data Processing Requirements in English", Working paper, Automatic Program generation Project, Moore School, University of Penna., 1976.

PRY77A Prywes, N.S., "Automatic Generation of Computer Programs", in Advances in Computers, Vol. 16, Academic Press, 1977.

PRY77B Prywes, N.S., "Automatic Generation of Computer Programs", Proceedings of NCC, 1977.

RAM73 Ramirez, J.A., "Automatic Generation of Data Conversion Programs using a Data Definition Language", Ph.D. Thesis, Moore School of Electrical Engineering, University of Pennsylvania, 1973.

RIN76 Rin, A.N., "Automatic Generation of Business Data Processing Programs from a Non-procedural Language", Ph.D. Dissertation, Moore School, University of Penna., 1976.

RUT75 Ruth, G., "The New Question Answerer", Internal Memo 21, Project MAC, MIT Cambridge, 1975.

RUT76A Ruth, G., "PROTOSYSTEM - I, An Automatic Programming System Prototype", Laboratory for

Computer Science, MIT, Cambridge, 1976.

RUT76B Ruth, G., "Automating the Software System Development Process", MIT Laboratory for Computer Science, 1976.

SHA76 Shastry, S.K., and N. Prywes, "Tolerance of Incompleteness or Ambiguities in Specification Languages", Working Paper, Moore School of Electrical Engineering, 1976.

SHA77 Shastry, S. K., N. Prywes and A. Pnueli, "Non-procedural Computer Programming with MODEL", COMPSAC 77, pp 500-506, Nov. 1977.

SHA78 Shastry, S.K., A. Pnueli and N. Prywes, "Algorithms on Array Graphs", Working Paper, Moore School, University of Pennsylvania, 1978.

SUG76 Sigiura, N., T. Hayashi, M. Yasuoka, "An Automatic Program Design and Generation System for Mini-computer Based Data Communication Systems", OKI. Electric Industry Co. Ltd., Tokyo Japan, Jan 76.

TEI72 Teichroew, D., "A survey of Languages for Stating requirements for Computer Based Information Systems", Proc. FJCC, Vol. 41, 1972.

TIN77 Tinaztepe, C., "FITS - TOP PART OF NOPAL, A Computer Program to design test specifications in the NOPAL Language", Ph.D. Thesis in Computer and Information Sciences, University of Pennsylvania, 1977.

ZLO75A Zloof, M.M., "Query-By-Example", Proc. National Computer Conference, AFIPS Press, Vol. 44, 1975, pp 431-438.

ZLO75B Zloof, M.M., "Query-By-Example: The invocation and Definition of Tables and Forms", RC-5115, IBM T. J. Watson Research Center, Aug 75.

ZLO76 Zloof, M.M., "Query-By-Example - Operations on Hierarchical Data Bases", AFIPS Conf. Proc., NCC 44, pp 431-438, 1975.

ZLO77A Zloof, M. M., "Query-By-Example: A Data base Language", IBM Systems Journal, Vol. 16, No. 4, 1977.

ZLO77B Zloof, M.M., and S. Peter de Jong, "The System for Business Automation (SBA): Programming Language", Communications of ACM, Vol. 20, No. 6, June 77.